Titre: Title:	Matrices cellulaires reconfigurables en point flottant dédiées au traitement des signaux
Auteur: Author:	Nabil El Ghali
Date:	2011
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	El Ghali, N. (2011). Matrices cellulaires reconfigurables en point flottant dédiées au traitement des signaux [Mémoire de maîtrise, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/650/

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/650/
Directeurs de recherche: Advisors:	Michael J. Corinthios
Programme: Program:	génie électrique

UNIVERSITÉ DE MONTRÉAL

MATRICES CELLULAIRES RECONFIGURABLES EN POINT FLOTTANT DÉDIÉES AU TRAITEMENT DES SIGNAUX

NABIL EL GHALI

DÉPARTEMENT DE GÉNIE ÉLECTRIQUE

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION

DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES

(GÉNIE ÉLECTRIQUE)

AOÛT 2011

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

MATRICES CELLULAIRES RECONFIGURABLES EN POINT FLOTTANT DÉDIÉES AU TRAITEMENT DES SIGNAUX

Présenté par : EL GHALI Nabil

en vue de l'obtention du diplôme de : Maîtrise ès Sciences Appliquées

a été dûment accepté par le jury d'examen constitué de :

M. DAVID Jean Pierre, Ph. D., président

M. CORINTHIOS Michael J., Ph. D., membre et directeur de recherche

M. BRAULT Jean-Jules, Ph. D., membre

DÉDICACE

À mes parents et ma famille...

REMERCIEMENTS

Je tiens tout d'abord à exprimer ma gratitude envers le professeur Michael Corinthios, que je remercie de m'avoir encadré et offert cette idée de recherche comme sujet de mémoire. Je lui adresse aussi un grand merci pour m'avoir introduit à la publication d'articles scientifiques. Je remercie le professeur Corinthios pour l'excellence et le niveau très élevé de son enseignement qui repoussent sans cesse les frontières du savoir connu et permet ainsi aux étudiants de contribuer de manière significative à la recherche.

Mes remerciements vont aussi aux membres du jury, le professeur Jean Pierre David, et le professeur Jean-Jules Brault qui en ayant l'amabilité d'accepter d'évaluer ce travail m'ont fourni un enseignement précieux et de très haute qualité. Je tiens également à exprimer ma reconnaissance et mes remerciements au professeur Saydy Lahcen dont l'aide a été très précieuse et essentielle. Mes remerciements vont aussi au professeur Hoang Le-Huy, ainsi que le professeur Gilles Roy, le professeur Michel A. Duguay et le professeur Roland Malhamé.

J'adresse aussi mes remerciements à Nathalie Levesque, Ginette Desparois, Anne-Marie Bourret, Thomas Martinuzzo, Morgan Guitton et Marie-Hélène Dupuis. Je salue aussi le travail de tous les membres du personnel de l'École Polytechnique Montréal, les secrétaires et techniciens du département de Génie Électrique qui offrent quotidiennement aux étudiants un environnement de travail exceptionnel.

Mes remerciements vont aussi à mes proches Souad, Mustapha, Kais, Skander, Chiraz, Rania, Yasmine, Driss, Rayan, Sirine, Adnen.

RÉSUMÉ

Les processeurs scalaires sont majoritairement utilisés de nos jours, pour le traitement des signaux numériques, par comparaison aux processeurs matriciels qui offrent pourtant plus de vitesse de calcul due à leur architecture parallèle traitant de nombreuses données en temps réel. Il existe une multitude d'architectures de matrices cellulaires. Cependant la grande majorité est très spécialisée pour le calcul d'une ou deux fonctions de traitement de signaux et seuls quelques processeurs matriciels sont reconfigurables afin de traiter la plupart des fonctions de traitement de signaux.

Ce mémoire présente l'architecture d'un processeur matriciel construit à partir de cellules complexes de calcul appelé "Module de Traitement Universel" (UPM). Ce processeur peut servir comme un module de propriété intellectuelle (IP block) destiné à être utilisé dans un FPGA pour le traitement des signaux. Des mêmes matrices d'UPMs sont reconfigurées en vue d'effectuer la plupart des opérations de Traitement Numérique des Signaux DSP incluant des fonctions de filtrage adaptatif récursives ou non et des fonctions d'analyse spectrale. Ce processeur peut être reconfiguré pour appliquer diverses transformées, filtres adaptatifs, filtres en treillis, en générations de fonctions, corrélations et en calcul de fonctions récursives qui peuvent être exécutées à grande vitesse. Pour une plus grande précision la conception est faite de manière à traiter les données en arithmétique point flottant. Afin de permettre le calcul de fonctions récursives l'unité de traitement UPM est construite avec un module de contrôle de récursivité. En outre l'UPM est conçu de manière à être mis en cascade afin d'augmenter l'ordre des opérations de traitement.

La conception logicielle de matrice 2x2 UPMs et 6x4 UPMs, qui sont programmées en langage Verilog-HDL, est simulée et testée avec les mêmes cellules reconfigurées en plusieurs fonctions telles que le filtrage adaptatif, l'analyse spectrale et le calcul de fonctions récursives. La même matrice de cellules à été simulée sur Matlab *Simulink* sous différentes configurations.

Les tests montrent que ce processeur offre une certitude de calcul acceptable pour tous les algorithmes proposés et une vitesse de calcul qui est limitée uniquement par la fréquence de l'horloge point flottant qui est au maximum de 100 *MHz*. Cependant les matrices cellulaires offrent une de performance de vitesse qui croît avec le nombre de cellules et qui est supérieure à celle offerte par les processeurs scalaires, aux *Duo Core*s, et des performances du même ordre de grandeur que les processeurs DSP les plus récents.

ABSTRACT

Scalar processors are commonly used today in contrast with array processors which offer a higher computation speed due to their parallel architecture dealing with a great number of data in real time. Several cellular arrays architectures exist. However, the vast majority is highly specialized for the computation of one or two signal processing functions and only a few are reconfigurable to handle most of the of signal processing functions.

This thesis presents the architecture of an array processor constructed using building blocks which are complex computation cells named Universal Processing Module (UPM). This array processor may serve as an intellectual property (IP block) to be used in FPGA technology and dedicated to signal processing. The same UPMs matrices are reconfigured to perform most of digital signal processing DSP operations including adaptive recursive and non recursive filtering, and spectral analysis functions. This processor can be reconfigured in order to compute transforms, adaptive filters, lattice filters, function generations, correlations and recursive functions, all performed at high speed. For greater accuracy the processor is constructed in floating point arithmetic. In order to enable computation of recursive functions, the UPM is built with a recursion control module. This processing element can also be indefinitely with the intention to increase filtering order.

The software design of a 2x2 UPMs and a 6x4 UPMs arrays which is programmed in Verilog-HDL language, is simulated and tested using same cells reconfigured in order to compute DSP algorithms such as adaptive filtering, spectral analysis and recursive functions. The same matrix of cell is simulated on Matlab *Simulink* through different configuration.

The processor is tested with all proposed reconfigurations and offers an acceptable computing precision. The computational speed is reduced only by the floating point clock frequency which maximum is 100 *MHz*. However, cellular arrays speed performance increases with the number of cells and is superior to those offered by scalar processors, *Duo Core*s processors and at the same range of order with performances offered by recent DSP processors.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

1D	Une Dimension
2D	Deux dimensions
ALU	Arithmetic Logical Unit
ASIC	Application Spécific Integrated Circuit
CLB	Configurable Logic Block
CORDIC	Coordinate Rotation Digital Computer
DCT	Discrete Cosine Transform
DHT	Discrete Hartley Transform
DSP	Digital Signal Processing
GOPS	Giga Octet Par Seconde
HDL	Hardware Description Language
HD-TV	High Definition Television
IDCT	Inverse Discrete Cosine Transform
IEEE	Institute of Electrical and Electronics Engineers
IIR	Infinite Impulse Response
IQ	Inverse Quantization
FFT	Fast Fourier Transform
FHT	Fast Hartley Transform
FIR	Finite Impulse Response
FPGA	Field Gate Programmable Array
FPU	Floating Point Unit
FWT	Fast Walsh-Hadamard Transform

HTC	Hilbert Transform Correlator
LAB	Logic Array Block
LE	Logic Element
LMS	Least Mean Square
MAC	Multiply Accumulate
MC	Motion Compensation
MIMD	Multiple Instructions Multiple Data
MIPS	Million Instruction Per Second
MUSE	Multiple Sub-Nyquist Sampling Encoding
NTSC	National Television System Committee
OIOO	Ordered Input Ordered Output
PLL	Phase-Locked Loop
PE	Processing Element
RAM	Random Access Memory
RLS	Recursive Least Square
RTL	Register Transfert Level
SIMD	Single Instruction Multiple Data
UPE	Universal Processing Element
UPM	Universal Processing Module
VGA	Video Graphic Array
VPE	Vidéo Processing Element

LISTE DES ANNEXES

INTRODUCTION

La théorie du traitement de signal est basée sur une multitude de fonctions mathématiques et d'algorithmes qui sont utilisés dans des domaines comme le filtrage, le codage, la détection, l'analyse, l'enregistrement, la reconnaissance, la reproduction et bien d'autres [1]. Dans la majorité de ces applications, des processeurs numériques ou microprocesseurs sont utilisés.

Les microprocesseurs et les processeurs scalaires utilisent pour leurs calculs des ALUs (Unité Arithmétique Logique) ou des FPUs (Unité Point Flottant) qui sont des circuits logiques utilisés pour des opérations simples telles que la multiplication, l'addition, l'inversion et d'autres calculs basiques. Néanmoins l'utilisation des microprocesseurs implique que ces fonctions mathématiques soient programmées et compilées de manière séquentielle, avec un certain nombre d'instructions, ce qui apporte un délai d'exécution. Les processeurs superscalaires utilisent quelques ALUs et FPUs additionnels par rapport aux processeurs scalaires, ce qui leurs donne une plus grande vitesse d'exécution des instructions et donc une plus grande vitesse de calcul. Il existe aussi des processeurs vectoriels qui sont plus rapides car ils traitent plusieurs données en parallèle et en temps réel. Cependant ces processeurs vectoriels sont trop spécialisés et n'offrent en général qu'une ou deux fonctions de traitement de signal.

Ce mémoire souligne les concepts détaillés qui ont été mis en œuvre dans la recherche et la conception d'un module de propriété intellectuelle (IP block) destiné à être utilisé dans un FPGA pour le traitement numérique des signaux. Ce module appelé UPM (Module Universel de Traitement) est dédié aux processeurs vectoriels ou matrices cellulaires et leurs permettent d'être reconfigurables en vue d'offrir une multitude d'opérations de traitement en parallèle et en point flottant pour avoir une grande précision de calcul.

CHAPITRE 1 REVUE DE LITTÉRATURE ET PROPOSITION DE RECHERCHE

Cette revue de littérature porte sur les différents processeurs scalaires et vectoriels dediés au Traitement Numérique des Signaux. La question de recherche, l'objectif général et les objectifs spécifiques y sont énoncés ainsi que les hypothèses scientifiques originales de recherche. La méthodologie de recherche employée y est décrite ainsi que les phases du projet et son échéancier.

1.1 Revue de littérature

Les processeurs scalaires et vectoriels spécialisés dédiés au traitement numérique des signaux qui ont fait l'objet de recherches et qui sont actuellement employés sont exposés dans cette section. Néanmoins il sera montré que le concept d'élément de traitement universel des signaux dédié aux processeurs vectoriels existe déjà.

1.1.1 Les processeurs Scalaires et Super Scalaires

Les processeurs scalaires peuvent être définis comme des processeurs qui traitent une ou deux données (en point fixe ou point flottant) par cycle d'horloge car ils ne contiennent généralement qu'une ou deux unités arithmétiques logiques ALUs et une unité point flottant FPU pour effectuer des calculs. Les processeurs scalaires ont fait l'objet de recherches qui ont abouti à la création de processeurs superscalaires qui sont constitués d'au moins quatre ALUs et deux FPUs permettant ainsi d'augmenter le nombre d'instructions exécutées en parallèle. Ces processeurs sont majoritairement utilisés dans l'implémentation des algorithmes de traitement numérique des signaux DSP, tels que les processeurs *Pentium* d'Intel ou le *Nios* l'Altera [2-5]. En effet le processeur scalaire, du fait de son architecture qui traite d'un nombre limité de données à la fois et de sa nature séquentielle, permet la programmation de la majorité des algorithmes de DSP. D'autres processeurs scalaires sont plus spécifiques et spécialement utilisés pour les traitements

de signaux comme le *TMS320* de Texas Instruments et le processeur *SHARC* d'Analog devices [6,7].

1.1.2 Les Processeurs Vectoriels ou Matrices cellulaires spécialisées

Par opposition aux processeurs scalaires, les processeurs vectoriels (ou Matrices Cellulaires ou Matrices Systoliques) traitent un vecteur de données en parallèle offrant ainsi une plus grande vitesse de calcul par coup d'horloge. Les processeurs vectoriels sont apparus dans les années 1970s et ont formé la base de la plupart des supers ordinateurs jusqu'aux années 1980s et 1990s [8]. Le développement des processeurs scalaires a entrainé le déclin des processeurs vectoriels traditionnels. Cependant, la plupart des processeurs scalaires comprennent aujourd'hui des modules de traitements vectoriels tels que les processeurs à une instruction et multiple données SIMD qu'on peut retrouver dans les cartes de jeux vidéo, les accélérateurs graphiques, les communications spatiales, l'imagerie biomédicale et les applications nécessitant une haute vitesse de calcul, utilisant de multiples données en parallèle. La conception des processeurs vectoriels a pour point de départ la recherche d'idées de systèmes spécifiques à une application donnée [9]. En effet, les processeurs vectoriels, qui sont spécialisés pour une application donnée, ont attiré l'attention des chercheurs pour des implémentations d'une ou deux fonctions spécifiques de traitements numériques tels que le filtrage à Réponse Finie/Infinie FIR/IIR, la Transformée de Cosinus Discrète DCT/IDCT, la Transformée Discrète de Hartley DHT, la Transformée Discrète de Hilbert, le filtrage en Treillis, la Transformée de Fourier Rapide FFT, les fonctions de corrélation et bien d'autres.

- Le Filtrage FIR/IIR et la transformée DCT/IDCT

Plusieurs processeurs vectoriels [10,15] ont été conçus pour le filtrage des signaux numériques en une dimension et ont une architecture principalement constituées d'un vecteur de multiplieurs accumulateurs appelé MACet de registres. Les multiplieurs accumulateurs MACs sont aussi employés dans le filtrage adaptatif en deux dimensions 2D [10] qui est principalement utilisé dans le traitement numérique des images. Plus particulièrement, le filtrage FIR 2D [10] et [16,18], peut être implémenté avec un processeur vectoriel constitué de vecteurs ou matrices d'éléments de traitement (Processing Element) PE pour le filtrage des images. La figure 1-1 montre l'exemple d'une architecture de processeur dédié aux convolutions, corrélations et filtrage [10]. Des processeurs vectoriels ont été proposés pour des applications de traitement vidéo où des

cellules de traitement en point fixe sont mises en cascade indéfiniment pour effectuer des transformées de cosinus discrètes DCT/IDCT 2D [19,21].

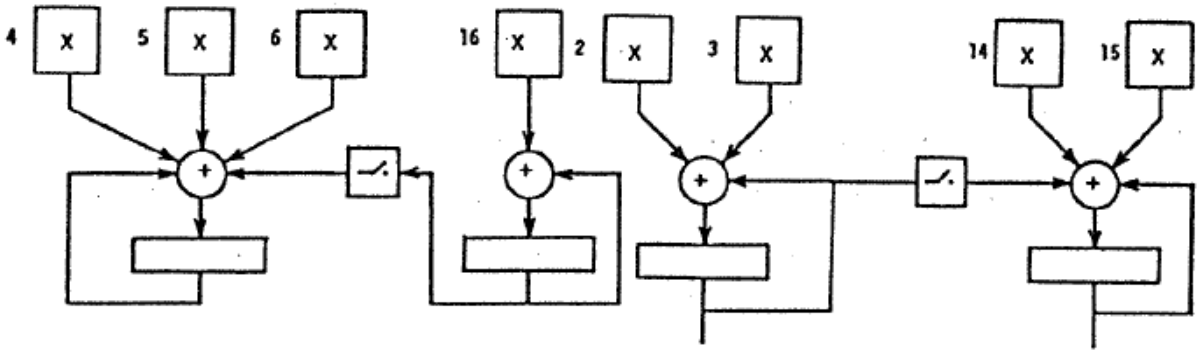


FIGURE 7

MAAP CELLS C_1, C_2, C_4 and C_{16}

FIGURE 8

MAAP CELLS C_8, C_3, C_{13} and C_{15}

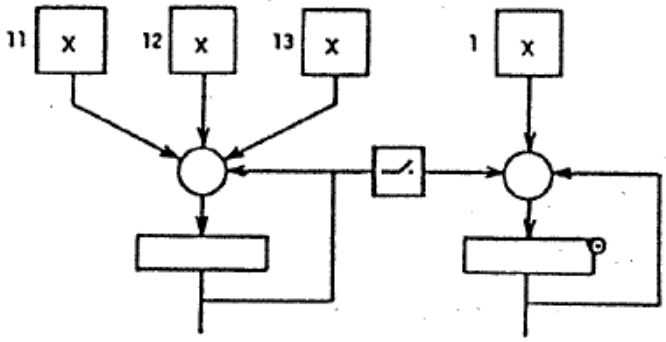


FIGURE 9

MAAP CELLS C_{11}, C_{12}, C_{13} and C_1

Figure 1-1 : Exemple de matrices d'UPE [10].

- Le Filtrage en treillis

Le filtrage en treillis est utilisé dans le traitement des signaux pour des applications telles que l'égalisation de canaux (channel equalization), l'identification de systèmes, les banques de filtrage et l'analyse de la voix et d'autres applications de filtrage [22,25]. Dans ces implémentations chaque cellule est caractérisée par la structure en treillis constituée de deux multiplieurs, deux additionneurs et d'au moins un registre de décalage temporel.

- La Transformée de Fourier rapide FFT et de Walsh-Hadamard FWT

La transformée de Fourier rapide FFT qui, permet une évaluation spectrale des signaux en temps réel, est très utilisée dans le traitement numérique des signaux. Des matrices de multiplieurs complexes et d'additionneurs complexes sont employées comme cellules pour effectuer une transformée de Fourier rapide FFT telles que les architectures proposées par le professeur Corinthios [10], [26,28]. Il est aussi possible d'utiliser des multiplieurs et additionneurs complexes afin de construire un processeur dédié aux transformées rapides de Walsh-Hadamard FWT [10], [29,30] en ordre naturel, dyadique et séquentiel mais aussi pour des transformées de Walsh généralisées telles que la transformée de Walsh-Paley Généralisée dont l'architecture est illustrée sur la figure 1-2 pour un ordre *N=27*.

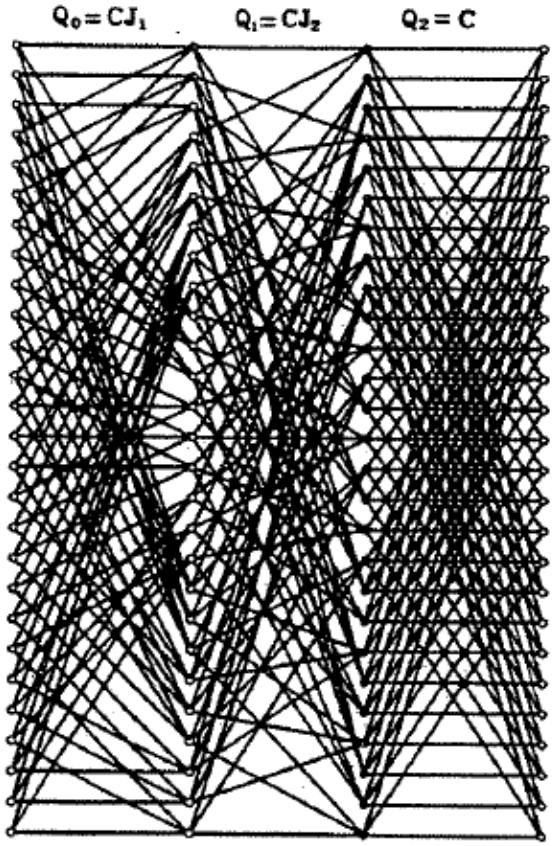


Fig. 1. Operators Q_0, Q_1 and Q_2 of the factorization of the GWP transformation matrix with $N = 27$, $p = 3$.

Figure 1-2 : Architecture d'une transformée de Walsh-Paley Généralisée ordre, *N*=27, [29].

- La Transformée de Hilbert et de Hankel

Des processeurs vectoriels sont aussi utilisés pour effectuer des transformées de Hilbert HTC dans lesquels chaque vecteur est constitué d'un multiplieur, un additionneur, une série de registre à décalage et de diviseurs [31,33].

- La Transformée de Hartley

Plusieurs études récentes ont été portées sur l'implémentation de la transformée de Hartley Rapide FHT sur des processeurs vectoriels appelés aussi circuits systoliques [34,39]. Dans la plupart de ces réalisations les matrices ou vecteurs d'éléments de traitements sont constitués d'additionneurs, multiplieurs de registre et d'un module de calcul numérique de rotation des coordonnées CORDIC qui sert au calcul des fonctions trigonométriques et hyperboliques.

- La séquence d'autocorrélation

Pour l'évaluation de la séquence d'autocorrélation des vecteurs de multiplieurs accumulateurs sont utilisés [40,41].

1.1.3 Les Matrices Cellulaires multifonctionnelles

- Le concept d'Élément de Traitement Universel (UPE)

L'idée d'un élément de traitement PE universel a déjà été mentionnée pour la première fois par le Professeur Corinthios qui, dans ses travaux [10] datant de 1985 et dans son article [29] de 1994, est le premier à introduire le concept d'un élément de traitement universel qui pourrait être mis en cascade afin de calculer plusieurs algorithmes de traitement de signaux avec une très grande vitesse de calcul. En effet, dans ce chapitre [10], après avoir discuté des algorithmes à géométrie constante pour l'analyse spectrale généralisée, le Professeur Corinthios propose un modèle d'unité universelle de traitement appelé UPE dont le schéma est montré sur les figure 1-3 et 1-4, capable de traiter des algorithmes qui peuvent être vus comme une multiplication matricielle (convolution, filtrage, corrélation). Cet UPE correspond à un arbre d'additionneur 3D, [10] p 2, qui pourrait incorporer des multiplieurs et additionneurs point flottant, [10] p 47. Les matrices cellulaires d'UPEs pourraient possiblement effectuer des calculs matriciels des multiplications accumulations et inversions de matrices pxp [10] p 47 et incluraient un réseau de permutation et de mémoires externes partitionnées, le tout destiné à l'analyse spectrale et au filtrage FIR et IIR.

En outre l'UPE est un concept d'architecture pour une cellule universelle de traitement contrairement à l'UPM conçu dans ce mémoire sous forme d'un module de propriété intellectuelle (IP block) destiné à être utilisé dans un FPGA.

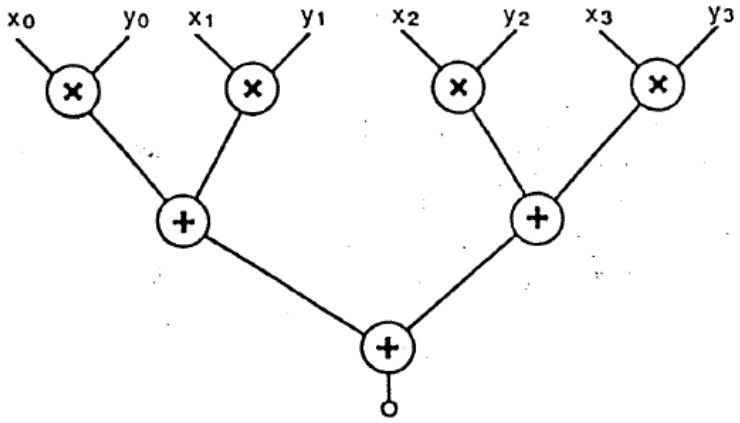


Fig. 8 (a) Side view of UPE and TATE, (b) A configuration example of UPE

Figure 1-3 : Exemple de configuration de l'UPE [10].

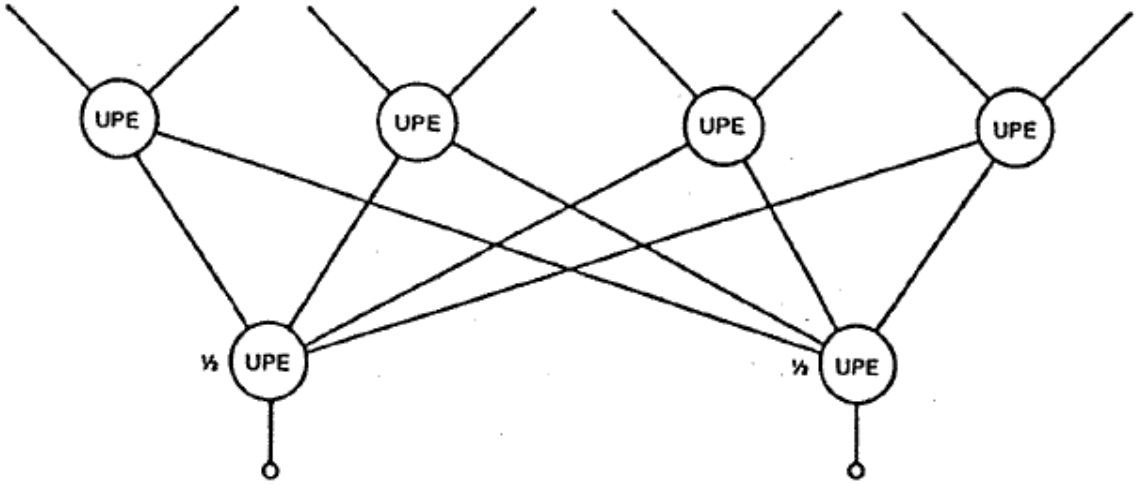


Fig. 9 Transform kernel using UPE's

Figure 1-4 : Configuration de matrices d'UPEs en DFT/Cherstenson [10].

- Brevets :

Des brevets ont été émis pour des processeurs vectoriels spécialisés tels que [42,46] qui montrent les types de processeurs les plus rencontrés dans l'industrie du traitement des signaux, effectuant des FFT, des multiplications accumulations et des opérateurs "*butterfly*".

1.2 Question de recherche énoncée

Est-il possible de concevoir un Module de Traitement des signaux multifonctionnel, reconfigurable par modification des connexions entre les mêmes cellules afin de calculer la plupart des fonctions de traitement des signaux tel que le filtrage FIR/IIR, le filtrage en treillis et les transformations spectrales? Plus particulièrement, pouvant être employés dans la construction de matrices cellulaires en point flottant dédiées à l'analyse spectrale, au filtrage FIR et IIR, au filtrage en treillis, à l'arithmétique complexe, aux calculs d'algorithmes récursifs, qui pourrait être mis en cascade afin d'augmenter l'ordre et la vitesse des opérations de traitement des signaux?

1.3 Objectif général

Concevoir théoriquement et tester sur FPGA (Field Programmable Gate Array) un Module Universel de Traitement UPM et les matrices cellulaires associées, en point flottant reconfigurables pour effectuer une multitude de fonctions DSP avec une architecture servant simultanément d'analyse spectrale, de filtrage adaptatif.

1.4 Objectifs spécifiques

- **Circuit numérique reconfigurable multifonctionnel**

Concevoir théoriquement un circuit numérique, appelé "*Module Universel de Traitement*" ou "*Cellule* ", qui effectue les opérations de traitement suivantes [1]:

  - Le Filtrage à réponse finie FIR.
  - Le Filtrage à réponse infinie IIR.
  - La Multiplication complexe.
  - La Transformée Discrète de Hilbert.
  - La Transformée Discrète Hankel.

- La Transformée Discrète de Hartley.
- La Transformée de Cosinus Discrète.
- La Corrélation et une Intercorrelation.
- Le Filtrage en treillis All-Pole, All-Zero et Zero-Pole.
- La Génération de fonctions trigonométriques, exponentielles, logarithmiques, Gamma et Bessel.
- La Transformée FFT.
- La Transformée de Walsh-Hadamard.
- La fonction Division utilisant l'algorithme de Newton-Raphson.
- La fonction racine carré utilisant l'algorithme de Newton-Raphson.

- **Matrices cellulaires**

Concevoir le module de traitement universel UPM pouvant être mis en cascade indéfiniment afin d'augmenter l'ordre de l'opération de traitement et de telle sorte qu'il puisse former un processeur Vectoriel ou des Matrices Cellulaires.

- **Point Flottant**

Concevoir l'Élément de Traitement de manière à ce qu'il effectue des calculs en point flottant afin d'augmenter la précision.

- **Programmation Test et Implémentation**

Programmer, tester et implémenter l'UPM sur circuit FPGA. Programmer et simuler des matrices cellulaires.

1.5 Hypothèses Scientifiques Originales de Recherche

La recherche proposée ici présente plusieurs hypothèses de contributions en termes de vitesse d'exécution, de nombre de fonctions offertes et en termes de précision:

- *Hypothèse 1:*

Si les matrices d'UPMs sont reconfigurables, offriraient-elles un très grand nombre d'algorithmes de traitement des signaux? Réfutabilité: Si trop fonctionnalités sont incluses, il se pourrait que le circuit numérique prenne trop d'espace logique.

- *Hypothèse 2:*

Si ce même module UPM pouvait être mis en cascade, cela permettrait-il d'augmenter le nombre d'opérations effectuées en un seul coup d'horloge et ainsi cela donnerait-il plus de parallélisme et donc plus de vitesse exprimée en *MHz* ? Réfutabilité: Si un grand nombre d'UPM est mis en cascade, il se pourrait que cela prenne trop d'espace logique.

- *Hypothèse 3:*

Si le format point flottant IEEE 754 est utilisé les calculs seront-ils beaucoup plus précis ?

Réfutabilité : les opérateurs en point flottant prennent beaucoup d'espace et souffrent d'une latence de plusieurs cycles d'horloge, ce qui est problématique lorsqu'on ne peut pas utiliser le pipeline. La réalisation de cette cellule en point fixe donnerait une latence plus faible et ferait gagner de l'espace logique.

1.6 Méthodologie employée

1.6.1 Phases du projet

a) Phase 1: Activités théoriques

1- Conception du circuit de filtrage FIR: Étude théorique des équations de filtrage FIR, IIR et schématisation du circuit numérique à partir des équations et en utilisant des multiplieurs et des additionneurs [1].
2- Conception du circuit Multiplieur Complexe: Étude théorique de l'équation Multiplieur complexe et schématisation du circuit à partir de l'équation et en utilisant le circuit précédent.
3- Ajout du Filtrage en Treillis All-Pole, All-Zero et Zero-Pole.
4- Ajout des fonctionnalités Corrélations et d'Intercorrelations.
5- Ajout de la Transformée Discrète de Hilbert.
6- Ajout de la Transformée Discrète de Hartley.
7- Ajout de la Transformée Discrète de Cosinus.
8- Ajout des Générations de fonctions trigonométriques, exponentielles, logarithmiques, Gamma el Bessel.
9- Ajout de la Transformée Discrète de Hankel.
10-Ajout de la fonction division.

11-Ajout de la fonction racine carré.

b) Phase 2: Programmation et implémentation sur FPGA

Programmation, simulation et implémentation de la Cellule sur FPGA en utilisant une carte de développement FPGA *Cyclone II* d'*Altera* et en utilisant l'environnement de développement logiciel *Quartus v9.1*et *Quartus v10.1*. Programmation et simulation de matrices d'UPMs sous différentes reconfigurations.

c) Phase 3: Test des sous-systèmes et du système.

1- Test par simulation des sous-fonctions de l'UPM en utilisant le Simulateur de *Quartus II*.
2- Test par simulation de l'UPM et de vecteurs d'UPMs sous différentes configurations.
3- Obtenir une estimation des performances
4- Implémentation de l'UPM sur FPGA et test avec écran VGA.

d) Phase 4: Rédaction d'article

Rédaction d'un article publié à une conférence.

e) Phase 5: Rédaction de mémoire

1- Rédaction de mémoire.
2- Soumission et présentation du mémoire.

1.6.2 Méthodologie utilisée

La méthodologie employée ici consiste à étudier, en premier lieu, les équations théoriques des opérations de traitement de signal à concevoir. Puis il s'agit ensuite de schématiser ces équations sous forme de circuits logiques à l'aide de multiplieurs et additionneurs point flottant. Pour se faire un filtre FIR/IIR et un multiplieur complexe sont conçus théoriquement. Les deux opérations sont ensuite rassemblées dans un seul module. De la même manière, les autres fonctionnalités sont schématisées à partir de leurs équations en modifiant le module afin qu'il effectue chaque fonctionnalité en prenant soin de minimiser l'espace pris par l'UPM et aussi en rajoutant du parallélisme afin de gagner de la vitesse.

Sur le plan pratique, l'architecture de l'UPM est programmée sur FPGA en Verilog-HDL en utilisant l'environnement de programmation *Quartus II* d'*Altera*. Des simulations, de plusieurs configurations de l'UPM et de vecteurs d'UPMs, sont réalisées par l'intermédiaire du simulateur

de *Quartus II*. La carte de test *Cyclone II* d'*Altera* est utilisée pour effectuer l'implémentation sur FPGA de l'UPM dont les résultats de tests sont affichés sur écran VGA.

1.7 Échéancier détaillé

Le diagramme de Gantt dans l'ANNEXE 1 décrit l'échéancier détaillé du projet.

CHAPITRE 2 ÉTUDE ET CONCEPTION THÉORIQUE

Les conceptions théoriques d'un élément de traitement, et les matrices cellulaires associées, reconfigurables dans le but d'effectuer la plupart des calculs de fonctions DSP en utilisant les mêmes cellules, sont présentées dans ce chapitre. À cet effet un module de contrôle de la récursivité, l'UPM et des matrices d'UPMs sont construits de telle manière à ce que les mêmes UPMs puissent être réutilisées et reconfigurées en filtres à Réponse Impulsionnelle Finie FIR, filtres à Réponse Impulsionnelle Infinie IIR. Pour se faire un module qui sert au calcul des fonctions récursives est construit et intégré à deux arbres de multiplieurs accumulateurs. D'autres reconfigurations permettent à ces mêmes cellules de fonctionner en mode filtre en treillis All-Pole, All-Zero et Pole-Zero. Des matrices de corrélations sont montrées comme des possibles implémentations en utilisant ces mêmes éléments de traitement. L'architecture parallèle de ces matrices cellulaires mène vers une évaluation rapide et efficace de la plupart des transformées telle que la transformée discrète de Fourier, la transformée discrète de Hilbert, la transformée discrète de Hartley, la transformée discrète de cosinus et la transformée discrète de Hankel [1]. La transformée rapide de Fourier, rapide de Walsh-Hadamard, rapide généralisée de Walsh-Hadamard et d'autres analyses spectrales généralisées sont aussi implémentées ainsi que d'autres réalisations comprenant la génération de fonctions qui emploie les polynômes de Chebyshev pour générer des fonctions trigonométriques, trigonométriques inverses, exponentielles, logarithmiques, Gamma et Bessel. La technique récursive de Newton-Raphson est utilisée pour construire des matrices cellulaires effectuant simultanément plusieurs itérations des fonctions division et racine carrée [1].

2.1 Le Filtre à Réponse Impulsionnelle Finie (FIR)

D'après le livre "*Signals, Systems, Transforms and Digital Signal Processing with MATLAB*"[1], le Filtre à Réponse Impulsionnelle Finie FIR est un filtre numérique qui a une réponse impulsionnelle nulle entre les intervalles de chaque prise d'échantillons dont le nombre est fini.

[1] Voir la Référence [1] p374.

Le filtre FIR est caractérisé par une réponse $y[n]$ basée uniquement sur les valeurs du signal d'entrée $x[n]$, il est donc non récursif. D'autre part, la forme temporelle du filtre peut être vue comme la convolution du signal d'entrée $x[n]$ avec les coefficients b_k, telle que représenté dans l'équation suivante :

$$FIR[n] = y[n] = \sum_{k=0}^{N-1} b_k x[n-k] \tag{2.1}$$

Cette équation décrit le filtrage FIR, qui est similaire à celle représentant la convolution temporelle. Le filtre FIR est aussi une moyenne pondérée de points. L'équation (2.1) peut être schématisée par la figure 2-1 a) dans laquelle la *séquence 0* du filtrage FIR a pour équation :

$$y[0] = b_0 x[0]$$

La figure 2-1 b) illustre la *séquence 1* qui s'écrit :

$$y[1] = b_0 x[1] + b_1 x[0]$$

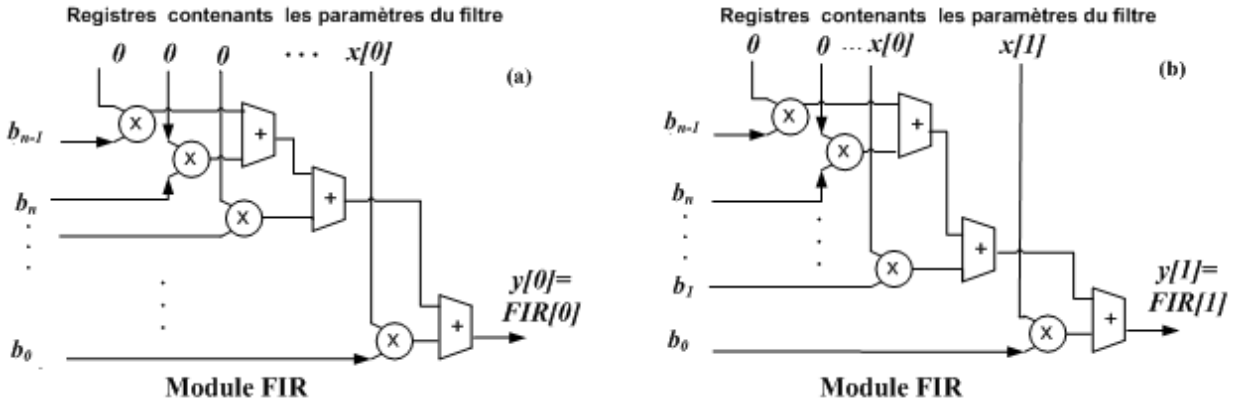


Figure 2-1 : Schéma théorique du filtrage FIR *séquence 0* a) et *séquence 1* b).

Des registres à décalage sont utilisés pour stocker les paramètres du filtre et les échantillons à filtrer. Cette figure montre la série de multiplieurs et d'additionneurs point flottant qui permet de faire une convolution de manière parallèle. De même, sur la figure 2-2, les *séquences 2* en a) et *n* en b), sont illustrées et peuvent être vues comme une convolution de données point flottant, décalées à chaque séquence. L'équation de la séquence *2* se formule comme suit :

$$y[2] = b_0 x[2] + b_1 x[1] + b_2 x[0]$$

De même que la *séquence n* s'écrit :

$$y[n] = \sum_{k=0}^{N-1} b_k x[n-k] \tag{2.1}$$

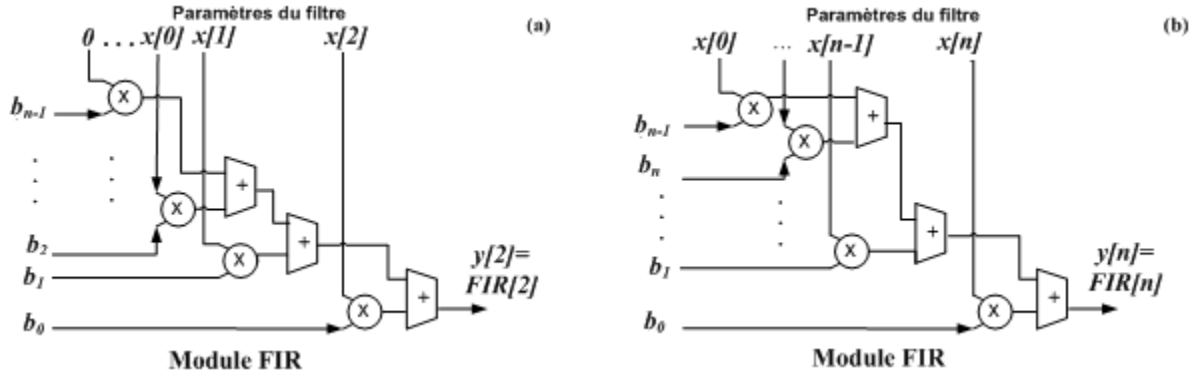


Figure 2-2 : Schéma théorique du filtrage FIR *séquence 2* a) et *séquence n* b).

2.2 Le Filtre à Réponse Impulsionnelle Infinie (IIR)

Le filtre numérique IIR peut être décrit par l'équation suivante, [1] p374 :

$$IIR[n] = y[n] = \sum_{k=0}^{M} b_k x[n-k] - \sum_{k=1}^{N} a_k y[n-k] \tag{2.2}$$

Ce filtre est caractérisé par une réponse $y[n]$ basée sur les valeurs du signal d'entrée $x[n]$ ainsi que des valeurs antérieures de cette même réponse $y[n-k]$, il est donc récursif. D'autre part, la forme temporelle du filtre est représentée par l'équation (2.2) dans laquelle le signal d'entrée est $x[n]$ et les coefficients b_k, sont les paramètres d'une convolution. Une autre convolution implique les paramètres a_k et la réponse antérieure $y[n-k]$. On remarque aussi que l'expression $\sum_{k=0}^{M} b_k x[n-k]$ équivaut à un filtrage FIR. Deux modules FIR peuvent donc être utilisés pour effectuer cette opération :

$$IIR[n] = y[n] = FIR[n] - \sum_{k=1}^{N} a_k y[n-k] \tag{2.2}$$

La figure 2-3 a) illustre la *séquence 0* dans laquelle deux modules de filtrage FIR sont utilisés d'une part pour le calcul de $FIR[0]$ et d'autre part pour la convolution de a_k et de $y[n-k]$ qui est soustraite à $FIR[0]$, ce qui donne le résultat du filtrage $IIR[0]$ en sortie.

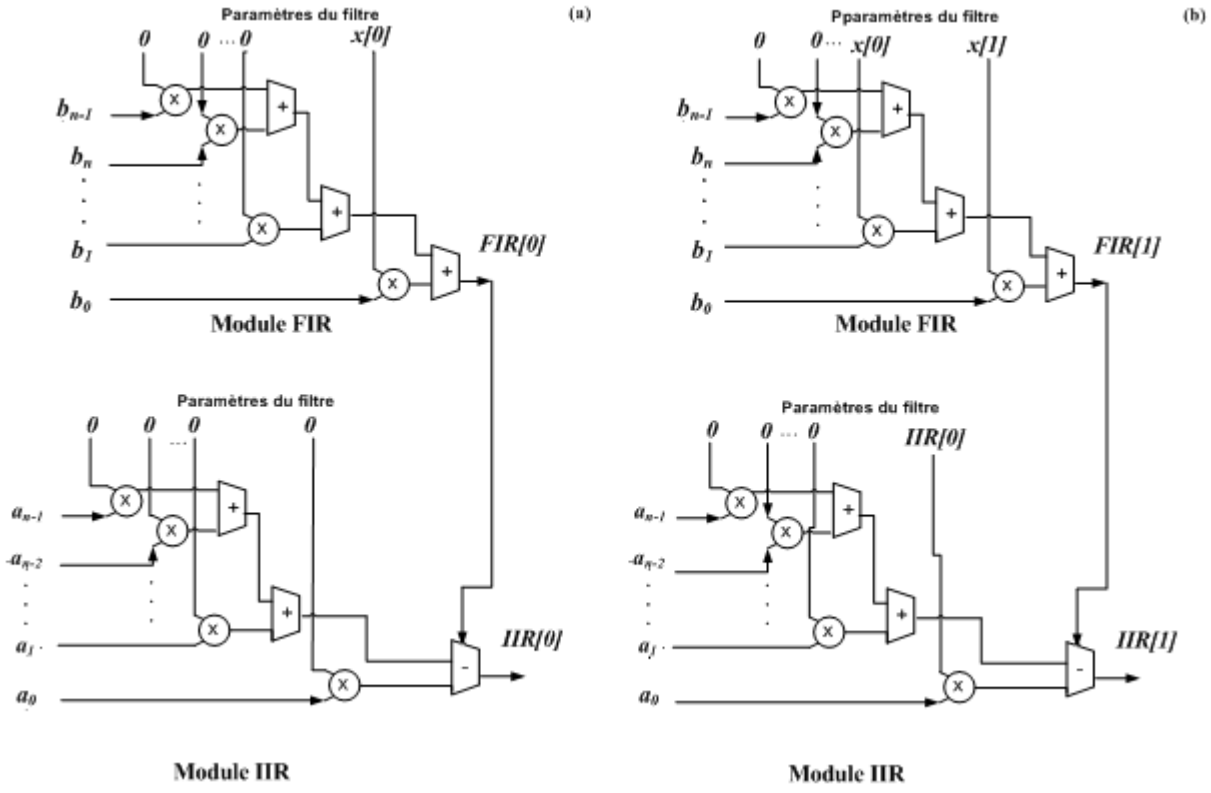


Figure 2-3 : Schéma théorique du filtrage IIR *séquence 0* a) et *séquence 1* b).

En reprenant l'équation (2.2), la *séquence 0* s'exprime de cette manière :

$$IIR[0] = y[0] = b_0 x[0] - 0 = FIR[0]$$

La *séquence 1*, qui est représentée sur la figure 2-3 b), s'écrit :

$$IIR[1] = y[1] = b_0 x[1] + b_1 x[0] - a_1 y[0] = FIR[1] - a_1 y[0]$$

La *séquence 2*, qui est illustrée sur la figure 2-4 a), peut être formulée de la sorte :

$$IIR[2] = y[2] = FIR[2] - a_1 y[1] - a_2 y[0]$$

La *séquence n*, sur la figure 2-4 b), s'écrit de cette façon:

$$y[n] = FIR[n] - \sum_{k=1}^{N} a_k y[n-k]$$

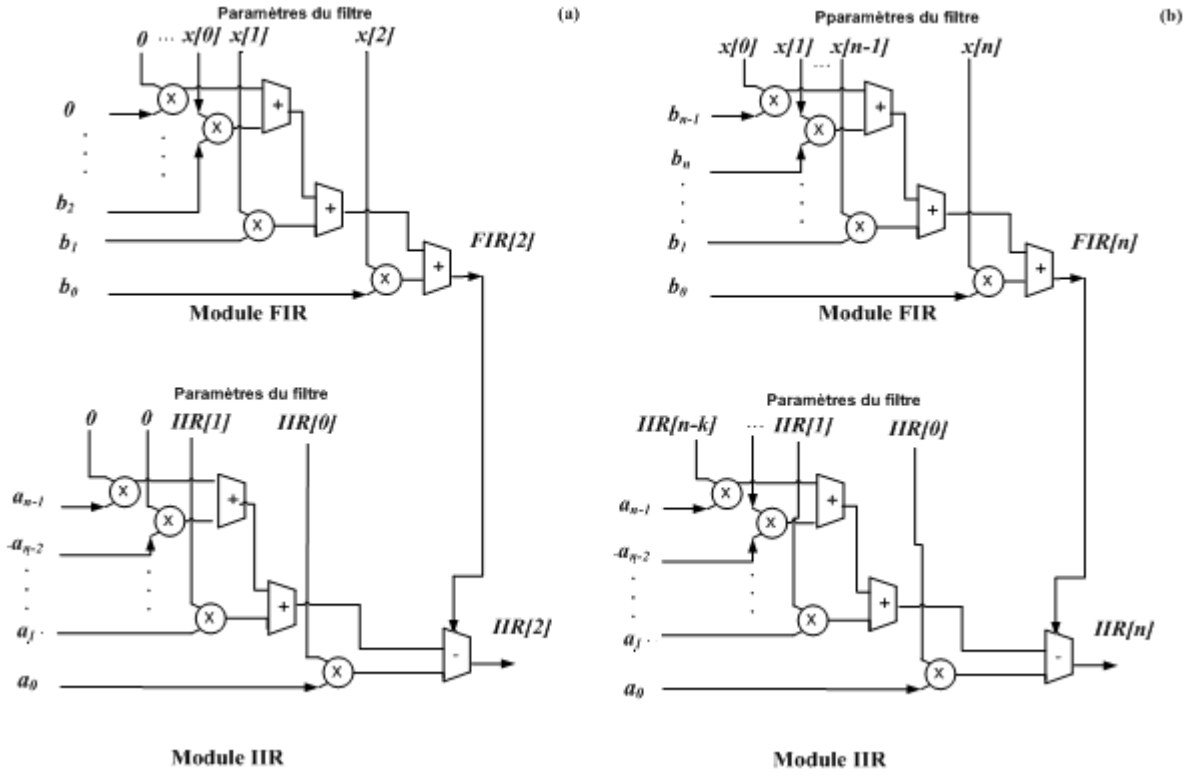


Figure 2-4 : Schéma théorique du filtrage IIR *séquence 2* a) et *séquence n* b).

2.3 Le Multiplieur et l'Additionneur Complexe

Les nombres complexes ont été imaginés et conçus au XVI^{ème} siècle par le mathématicien italien Gerolamo Cardano. Les règles de multiplication, d'addition, de soustraction et de division ont, par la suite, été établies par le mathématicien italien Rafael Bombelli au même siècle. Soit deux nombres complexes *A* et *B,* et *C* le résultat de leur multiplication qui est donné par l'équation (2.3).

$$\begin{aligned} &\left(\Re\{A\}+ j\Im\{A\}\right)*\left(\Re\{B\}+ j\Im\{B\}\right) \\ &= \Re\{A\}\Re\{B\}-\Im\{A\}\Im\{B\}+ j\left(\Re\{A\}\Im\{B\}+\Im\{A\}\Re\{B\}\right)=\Re\{C\}+ j\Im\{C\} \end{aligned} \tag{2.3}$$

Dans la figure 2-5, le module IIR est modifié de sorte qu'il effectue les opérations de multiplications complexes. Ainsi deux multiplications sont calculées en parallèle en une seule séquence, le module du haut donne les parties réelles des résultats tandis que celui du bas, les

parties imaginaires. Cette architecture peut être reconfigurée afin d'effectuer des additions et soustractions complexes.

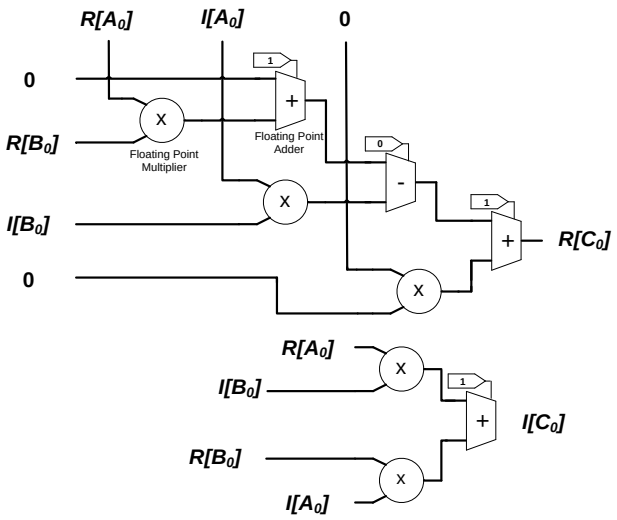


Figure 2-5 : Multiplication complexe

Afin de permettre à l'UPM d'effectuer des calculs récursifs et du filtrage IIR, un module, appelé *Module de Contrôle de Récursivité* est introduit. Ce module permet d'effectuer des calculs récursifs, de contrôler le choix entre la multiplication complexe et le filtrage IIR et de choisir la sélection de l'entrée IIR récursive. Ce module est montré dans la figure 2-6 dans laquelle les noms des entrées et sorties de ce module sont donnés en a) ainsi que sa configuration en multiplieur complexe en b). Ce module est construit avec trois multiplexeurs, trois registres à décalage et comprend quatre entrées de *32*-bits, trois entrées de contrôle de *1*-bit et trois sorties de *32*-bits. Sur cette figure l'entrée "*bit1*" sert à contrôler le choix entre le mode multiplieur complexe et filtrage, l'horloge "*clk_bit2*" sert à contrôler les registres à décalage et l'entrée "*bit3*" sert à sélectionner l'entrée IIR récursive.

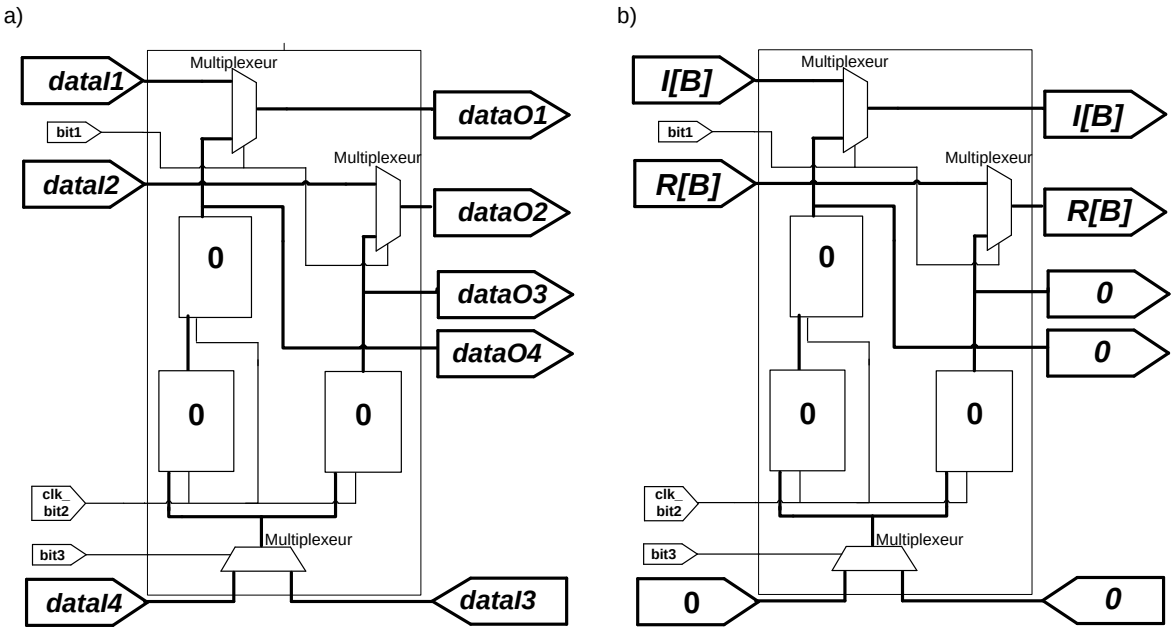


Figure 2-6 : Module de contrôle de récursivité a) noms des entrées et sorties b) configuration en multiplieur complexe.

La figure 2-7 a) illustre le module de contrôle de récursivité configuré en mode "Multiplication Complexe". La figure 2-7 b) indique la configuration du mode IIR pour la *séquence n*=0, la figure 2-7 c) pour *n*=1 et la figure 2-7 d) pour *n*=2. Le "*bit1*", qui contrôle deux multiplexeurs, permet la sélection entre les données complexes et les résultats passés du calcul IIR provenant des registres à décalage. Ces trois registres à décalage servent à enregistrer les valeurs passées du filtrage IIR. Si l'horloge "*clk_bit2*" de contrôle des registres à décalage est à *1*, les données sont enregistrées et utilisées comme entrées des deux multiplexeurs du haut. Un troisième multiplexeur, dans le bas de la figure, sert à sélectionner l'entrée IIR par l'intermédiaire du "*bit3*". Si ce bit est à *1*, le module prend comme entrée IIR les résultats provenant d'autres UPMs, et dans le cas contraire le module prend comme entrée IIR les résultats provenant du même UPM.

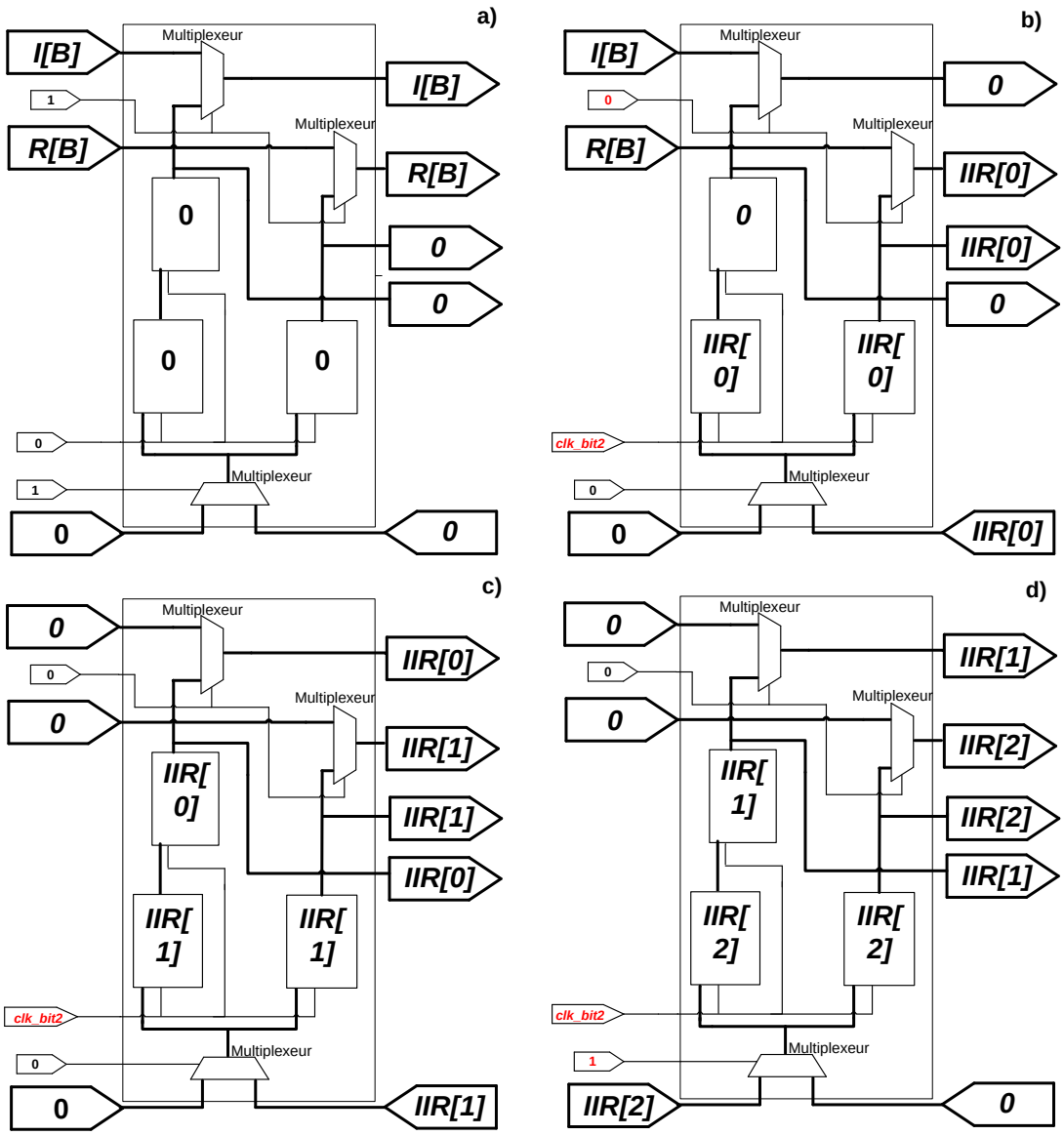


Figure 2-7 : Module de récursivité en mode Multiplication Complexe a) et Filtrage b), c), d).

2.4 Le Module Universel de Traitement (UPM)

Le Module Universel de Traitement, qui est schématisé dans la figure 2-8, est constitué de cinq multiplieurs, six additionneurs en point flottant et d'un module de contrôle de récursivité. Les additionneurs peuvent effectuer une soustraction grâce à un bit de contrôle du signe de l'opération représenté par une entrée avec un *1* pour l'addition ou un *0* pour la soustraction. Les entrées de signes des additionneurs sont représentés de cette manière dans le schéma suivant.

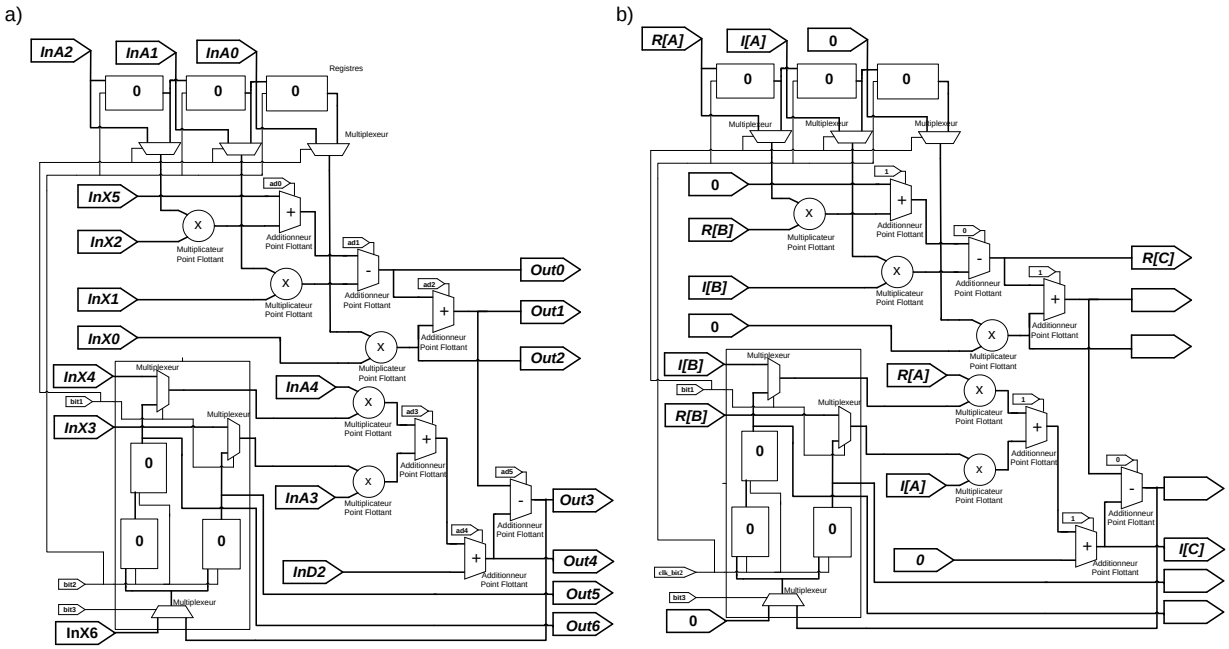


Figure 2-8 : a) Entrées et sorties de l'UPM b) UPM configuré en Multiplieur Complexe.

Cette cellule comprend treize entrées et sept sorties de données de *32*-bits, trois entrées de *1*-bit pour le contrôle du module de récursivité et six entrées de *1*-bit pour le contrôle des additionneurs. La partie supérieure de l'UPM est construite avec trois registres à décalage et trois multiplexeurs. Afin de permettre le fonctionnement en mode filtrage, une des entrées de la partie supérieure a été connectée à une série de registres à décalage qui sert à décaler temporellement les données provenant de cette entrée. Des multiplexeurs sont utilisés pour permettre à l'UPM de fonctionner en deux modes : filtrage et opération complexe.

Tel qu'illustré sur la figure 2-8, le "*bit1*" est mis à *1* afin de contrôler les données en entrée de telle sorte que l'UPM fonctionne en mode multiplieur complexe. Ce mode est aussi montré sur la figure 2-7 a). Dans cette configuration les calculs se font en une seule séquence et le résultat apparait en sortie. Les équations arithmétiques (2.4) à (2.10) du circuit UPM expriment chaque sortie, *Out0*, *Out1*, *Out2*, *Out3*, *Out4*, *Out5* et *Out6* en fonction des entrées de données, de *32*-bit, nommées sur la figure 2-8.

Soit la fonction *mux*(*test*,*X*,*Y*) qui retourne *X* lorsque *test* est vrai sinon elle retourne *Y*.

$$Out0 = mux\left(bit1,\ InA2 * Z^{-1},\ InA2\right) * InX2 + InX5 + mux\left(bit1,\ InA2 * Z^{-2},\ InA1\right) * InX1 \quad (2.4)$$

$$Out1 = Out0 + Out2 \quad (2.5)$$

$$Out2 = mux\left(bit1,\ InA2 * Z^{-3},\ InA0\right) * InX0 \quad (2.6)$$

$$Out3 = Out1 + Out4 \quad (2.7)$$

$$Out4 = mux\left(bit1,\ Out6,\ InX4\right) * InA4 + mux\left(bit1,\ Out5,\ InX3\right) * InA3 + InD2 \quad (2.8)$$

$$Out5 = mux(bit3,\ Out3,\ InX6) * Z^{-1} \quad (2.9)$$

$$Out6 = mux(bit3,\ Out3,\ InX6) * Z^{-2} \quad (2.10)$$

Le filtrage FIR/IIR est illustré sur la figure 2-9 pour les *séquences 0*, *séquences 1*, *séquences 2* et *séquences 3*. Dans ce mode, le bit "*bit1*" de contrôle des multiplexeurs est mis à zéro et l'horloge "*clk_bit2*" permet de contrôler les registres à décalage et ainsi de décaler temporellement les données. Le "*bit3*" peut être utilisé pour changer l'entrée l'IIR du module de récursivité comme illustré sur la figure 2-9 d). En effet cette entrée sert à mettre en cascade plusieurs UPMs afin d'augmenter l'ordre du filtrage tel que montrée sur les figures 2-10, 2-11, 2-12, 2-13 qui illustrent respectivement les *séquences 3, 4, 5* et *6* de deux UPMs configurés en cascade afin de faire du filtrage FIR/IIR *N=4*. De même, la figure 2-14 montre deux UPMs mis en cascade pour effectuer du filtrage FIR/IIR ordre *N=6, séquences 5* et *7*. Les équations arithmétiques (2.4) à (2.10) sont utilisées pour représenter un UPM configuré en filtrage FIR/IIR (Fig. 2-9) à l'aide des équations (2.11) à (2.17), en posant la valeur de l'entrée de données *InA2* de l'UPM égale à la valeur de

l'entrée du processeur matriciel "*Samples*". La valeur de *bit1* est mise égale à *0* en binaire et la valeur de *bit3* égale à *1* en binaire. Les valeurs respectives de *ad0*, *ad1*, *ad2*, *ad3*, *ad4*, *ad5* sont prises égales à *1*, *1*, *1*, *1*, *1* , *0* en binaire et les valeurs des entrées de données *InA0, InA1, InX5, InX3, InX4* et *InD2* égales à *0*, ce qui donne :

$$Out0 = InA2 * Z^{-1} * InX2 + InA2 * Z^{-2} * InX1 \tag{2.11}$$

$$Out1 = Out0 + Out2 \tag{2.12}$$

$$Out2 = InA2 * Z^{-3} * InX0 \tag{2.13}$$

$$Out3 = Out1 - Out4 \tag{2.14}$$

$$Out4 = Out6 * InA4 + Out5 * InA3 \tag{2.15}$$

$$Out5 = Out3 * Z^{-1} \tag{2.16}$$

$$Out6 = Out3 * Z^{-2} \tag{2.17}$$

Ces équations sont valables pour la (Fig. 2-9), cependant pour une matrice cellulaire comprenant *n* UPMs (Fig. 2-10 et Fig. 2-14), la sortie *Out3*, du n^{eme} UPM, est connectée à l'entrée *InX6* du premier UPM du vecteur, tandis que pour les autres cellules du vecteur les sorties *Out1*, *Out4*, *Out6* de chaque UPM sont respectivement connectées aux entrées *InX5*, *InD2*, *InX6* de l'UPM suivant dans le vecteur.

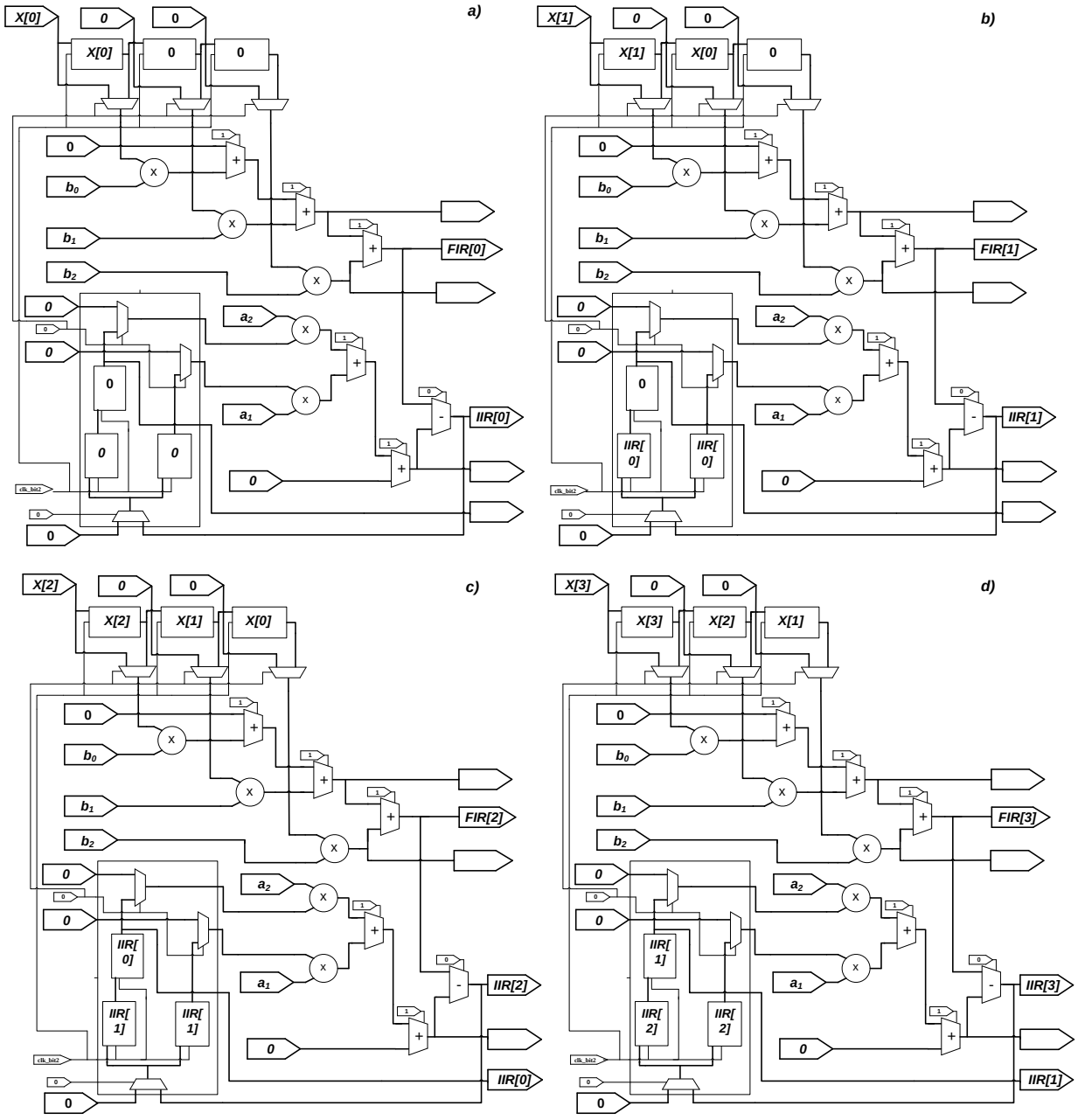


Figure 2-9 : UPM en mode FIR/IIR ordre *N*=2 *séquence 0* a), *séquence 1* b), *séquence 2* c) et *séquence 3* d).

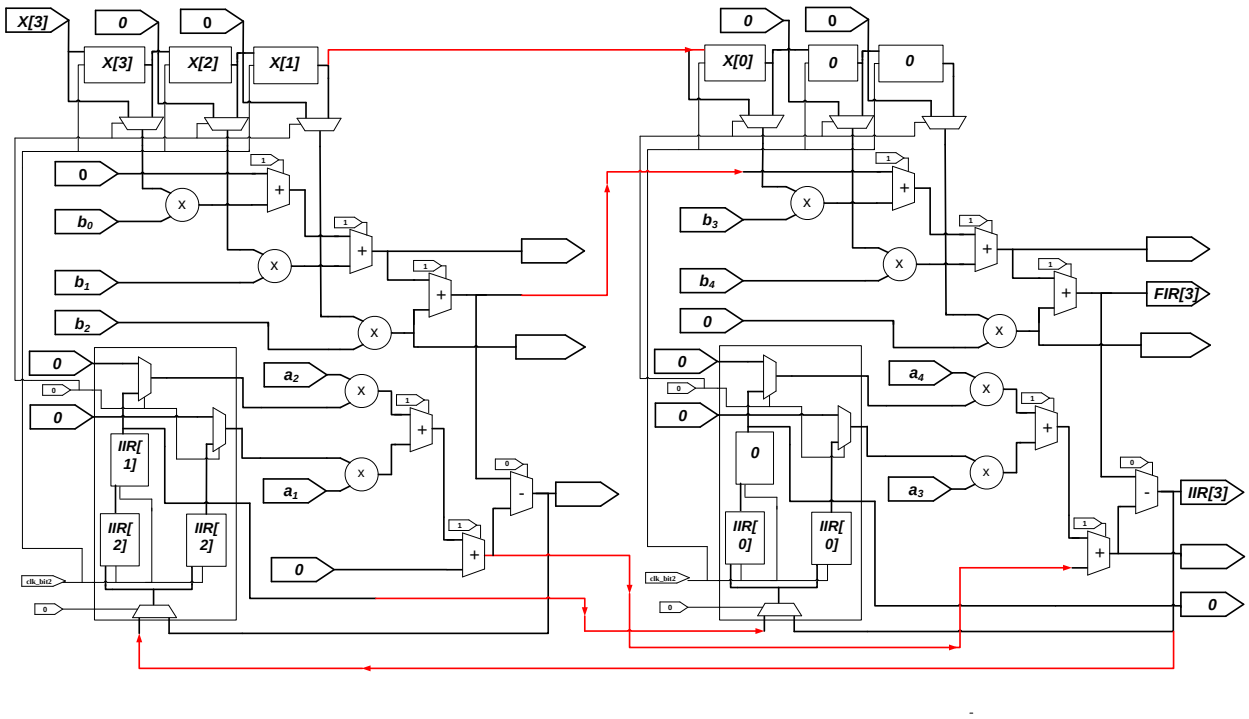


Figure 2-10 : Cascade de deux UPMs en mode filtrage FIR/IIR ordre *N=4*, *séquence 3*.

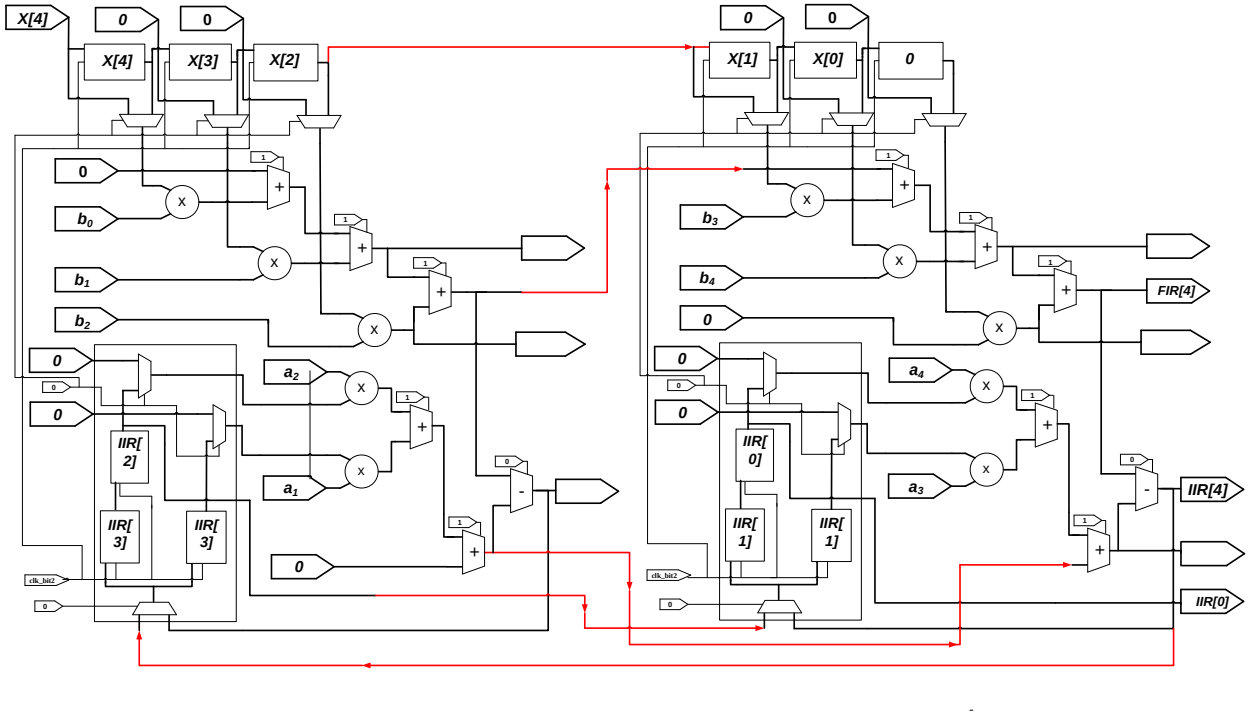


Figure 2-11 : Cascade de deux UPMs en mode filtrage FIR/IIR ordre *N=4*, *séquence 4*.

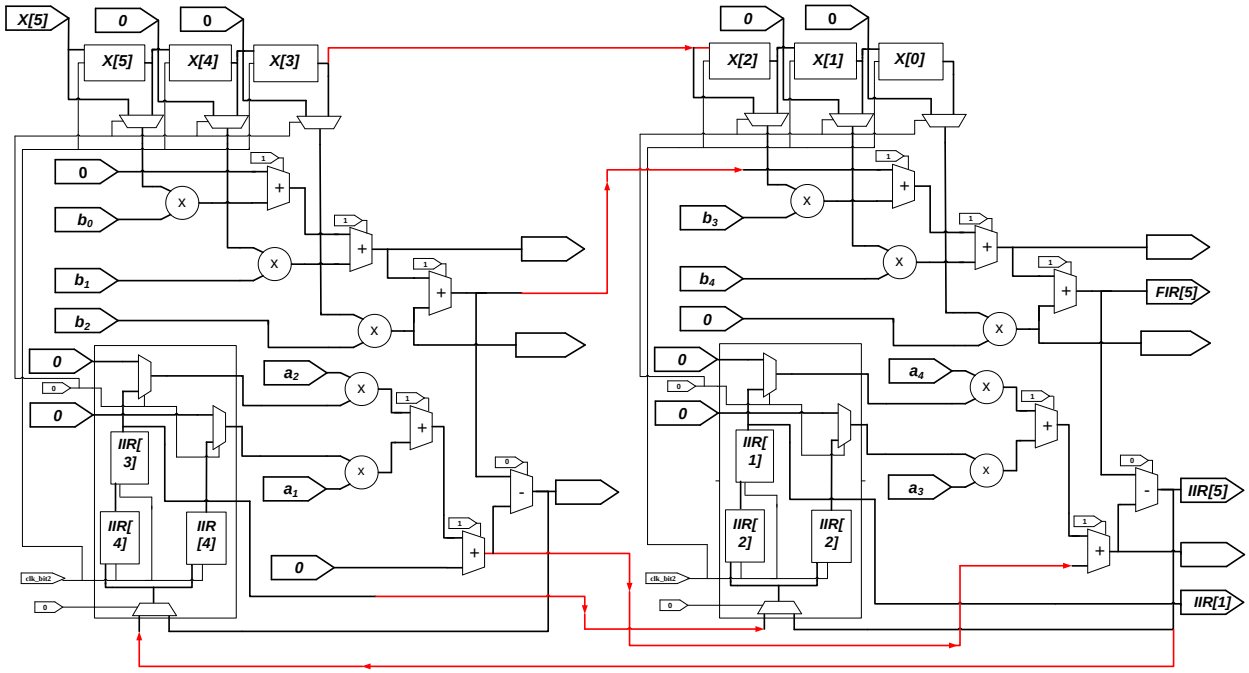


Figure 2-12 : Cascade de deux UPMs en mode filtrage FIR/IIR ordre *N=4*, *séquence 5*.

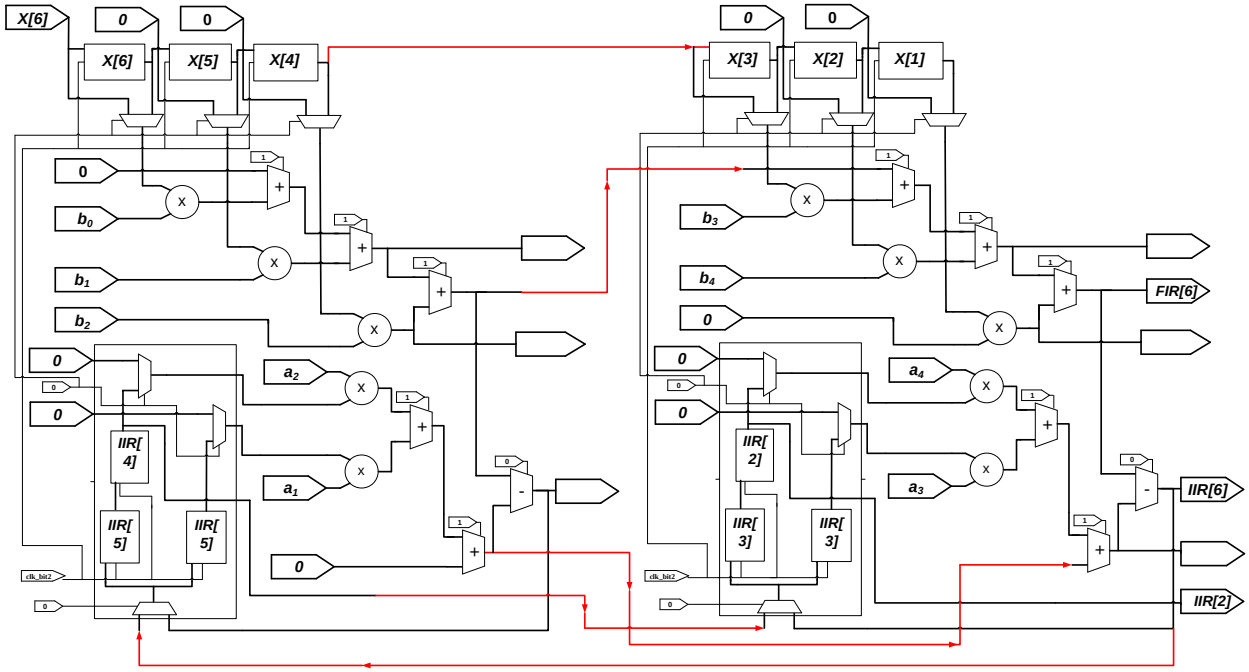


Figure 2-13 : Cascade de deux UPMs en mode filtrage FIR/IIR ordre *N=4*, *séquence 6*.

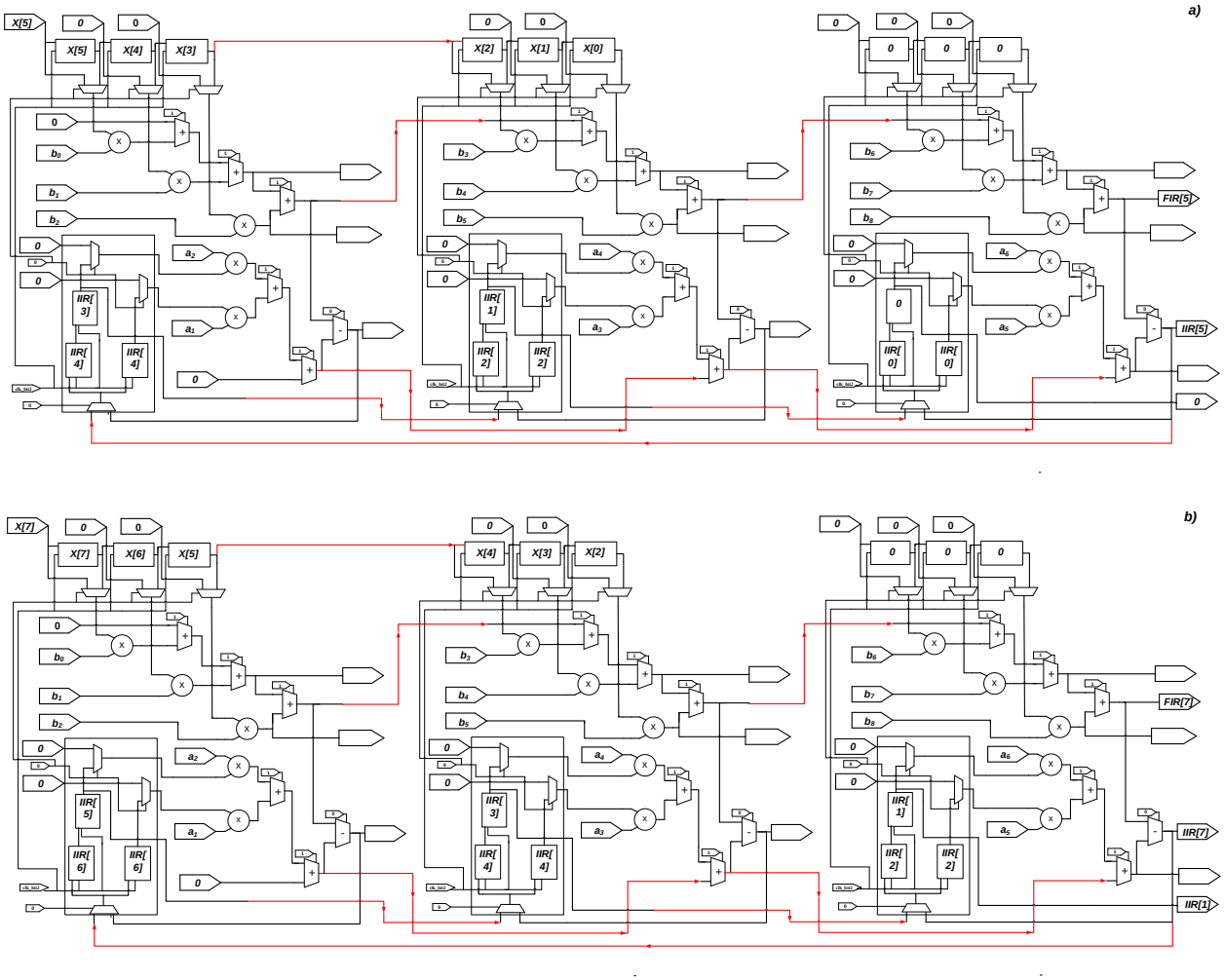


Figure 2-14 : Configuration filtrage FIR/IIR ordre *N=6*, *séquence 5* a) et *séquence 7* b).

2.5 Filtrage en Treillis

2.5.1 Le Filtre en Treillis All-Zero à un étage

Les filtres en treillis (Lattice Filters) sont très utilisés dans le traitement numérique vocal et dans l'implémentation des filtres adaptatifs, [1], p760-775. Le filtre en treillis All-Zero à un étage est décrit par les équations suivantes :

$$\begin{aligned} e_1[n] &= x[n] + k_1 x[n-1] \\ \tilde{e}_1[n] &= x[n-1] + k_1 x[n] \end{aligned} \tag{2.18}$$

La figure 2-15 illustre le schéma théorique d'un filtrage en treillis All-Zero à trois étages tiré de [1] p 767, et la figure 2-16 montre un UPM configuré en filtrage All-Zero à un étage, dans lequel $e_1[n]$et $\tilde{e}_1[n]$sont les sorties du filtre, $x[n]$ est la séquence à filtrer et k_1 le coefficient de réflexion du filtre. Les équations (2.4) à (2.10) peuvent être réécrites pour représenter un UPM configuré en filtrage All-Zero (Fig. 2-16) à l'aide des équations (2.19) à (2.25), en posant que les valeurs des entrées de données *InX2, InX3* de l'UPM sont égales à la valeur de l'entrée du processeur matriciel "*Samples*". Les valeurs des entrées de données *InA0, InX0, InD2* et *InX5* sont mises égales à *0* et les valeurs de *InX1, InX4* de l'UPM sont posées égales à la valeur de l'entrée du processeur matriciel "*Samples*"*Z^{-1} qui est retardée de *1*. La valeur de *bit1* est initialisée égale à *1* en binaire ainsi que la valeur de *bit3*. Les valeurs respectives de *ad0, ad1, ad2, ad3, ad4, ad5* sont mises égales à *1, 1, 1, 1, 1 , 1* en binaire. Pour un vecteur d'UPMs (Fig. 2-17 et 2-18) le premier UPM prend les mêmes valeurs de configuration discutées dans la phrase précédente. Cependant la sortie *Out4* de chaque UPM est connectée à l'entrée *InX6* du même UPM; de même que pour chaque UPM (excepté celle du premier UPM du vecteur) la sortie *Out0* est connectée aux entrées de données *InX2 et InX3* de l'UPM suivant, la sortie *Out5* de chaque UPM est connectée aux entrées *InX1* et *InX4* de l'UPM suivant :

$$Out0 = InA2 * Samples + InA1 * Samples * Z^{-1} \tag{2.19}$$

$$Out1 = Out0 + Out2 \tag{2.20}$$

$$Out2 = 0 \tag{2.21}$$

$$Out3 = Out1 + Out4 \tag{2.22}$$

$$Out4 = Samples * Z^{-1} * InA4 + Samples * InA3 \tag{2.23}$$

$$Out5 = Out4 * Z^{-1} \tag{2.24}$$

$$Out6 = Out4 * Z^{-2} \tag{2.25}$$

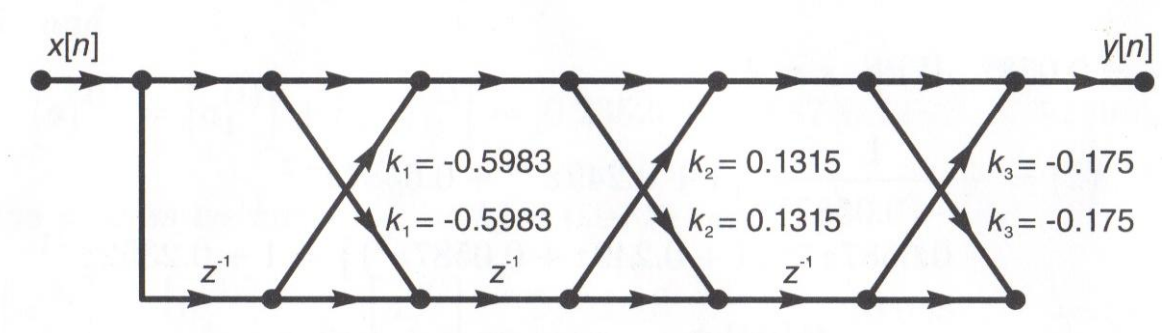


Figure 2-15 : Filtre en treillis All-Zero trois étages, [1] p 767.

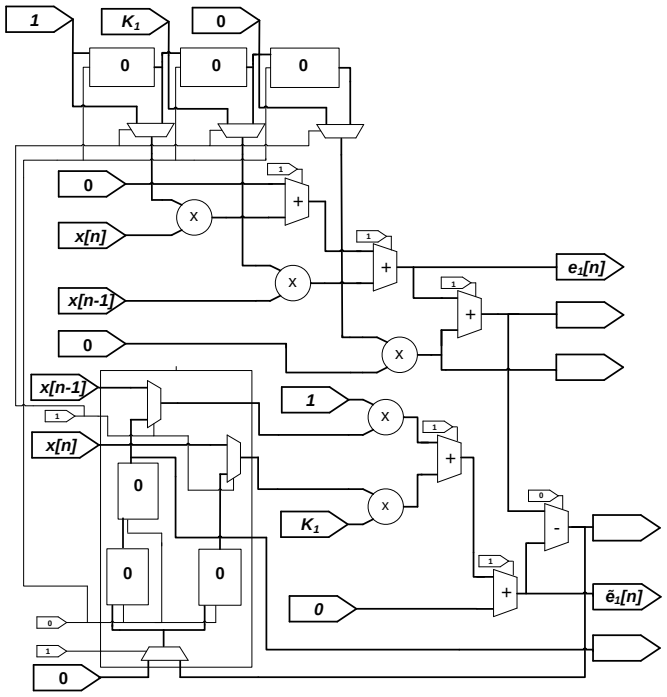


Figure 2-16 : UPM configurée en Filtre en Treillis All-Zero, un étage.

2.5.2 Les Filtres en Treillis All-Zero à deux étages et *s* étages

De la même manière que le filtre en treillis d'ordre *1*, le deuxième étage peut être représenté par l'expression suivante :

$$\begin{aligned} e_2[n] &= e_1[n] + k_2 \tilde{e}_1[n-1] \\ \tilde{e}_2[n] &= \tilde{e}_1[n-1] + k_2 e_1[n] \end{aligned} \tag{2.26}$$

Il est possible de mettre en cascade plusieurs filtres All-Zero de sorte que les entrées de l'étage *2*, $e_1[n]$ et $\tilde{e}_1[n]$, correspondent aux sorties de l'étage *1*. Les résultats $e_2[n]$et $\tilde{e}_2[n]$ sont les sorties du second étage et k_2 le coefficient de réflexion du second étage. L'équation du filtre All-Zero d'ordre général s'écrit comme en (2.27). Ce filtre est illustré sur la figure 2-17.

$$\begin{aligned} e_S[n] &= e_{S-1}[n] + k_S \tilde{e}_{S-1}[n-1] \\ \tilde{e}_S[n] &= \tilde{e}_{S-1}[n-1] + k_S e_{S-1}[n] \end{aligned} \tag{2.27}$$

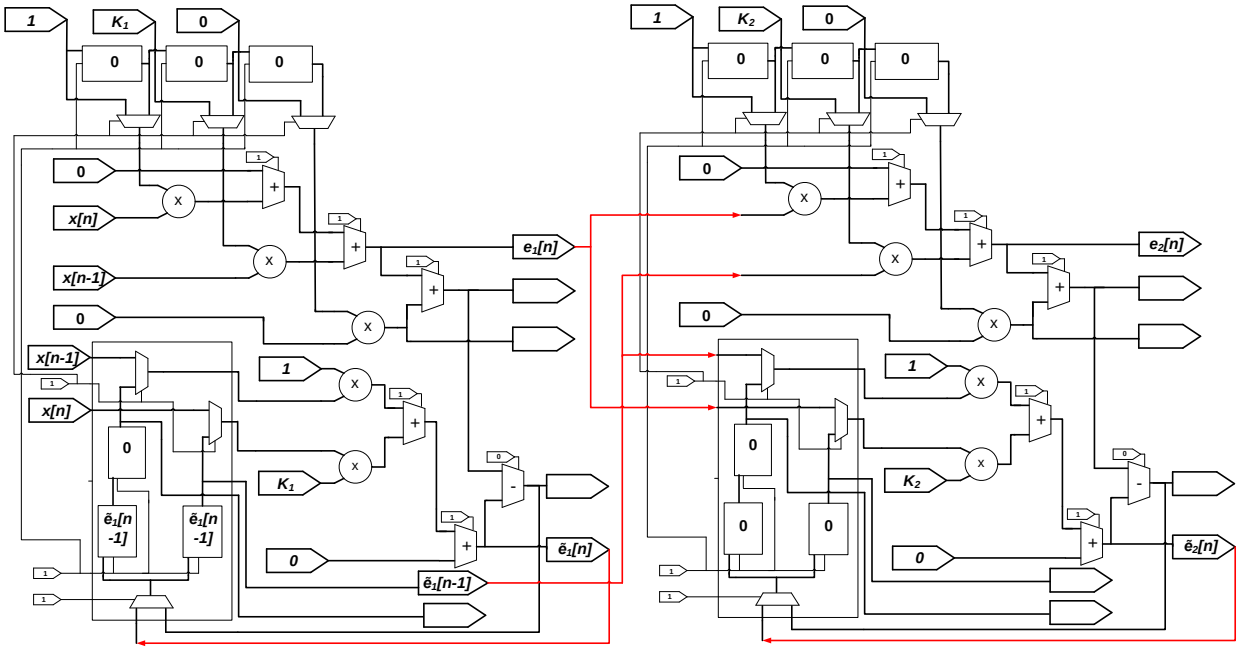


Figure 2-17 : UPM configurée en Filtre en Treillis All-Zero, deux étages.

La figure 2-18 représente le filtrage en treillis All-Zero à s étages. Tout comme les équations (2.18) et (2.26), l'équation (2.27) est récursive et dépend des valeurs passées de $\tilde{e}_S[n]$. Le module de récursivité est utilisé à cet effet, comme schématisé sur les figures 2-17 et 2-18. La valeur de $\tilde{e}_S[n]$ est mise en entrée dans le module de récursivité et un registre à décalage permet d'enregistrer $\tilde{e}_S[n-1]$. Une sortie transfère la valeur de $\tilde{e}_S[n-1]$vers l'entrée du deuxième étage du filtre en treillis. Il est ainsi possible de créer un vecteur de filtrage en treillis dont le nombre d'étage est très élevé.

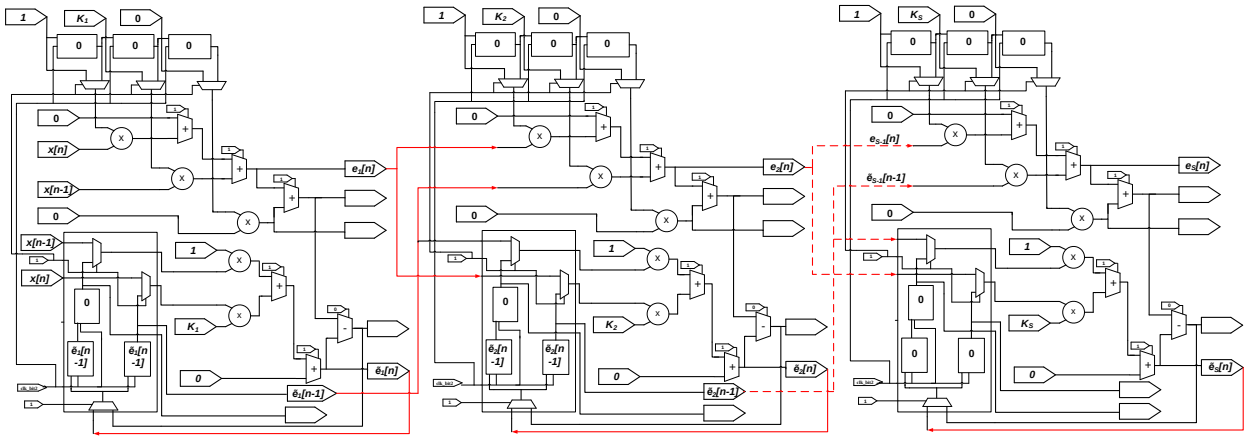


Figure 2-18 : Vecteur d'UPMs configurée en Filtre en Treillis All-Zero, *s* étages.

2.5.3 Le Filtre All-Pole

Le filtre en treillis All-Pole d'ordre général, [1] p770-773, a pour équation (2.28). À chaque $s^{ème}$ étage, la sortie $d_{S-1}[n]$ dépend du paramètre k_S, de l'entrée $d_S[n]$ et de $\tilde{d}_{S-1}[n\text{-}1]$ qui représente une valeur passé de $\tilde{d}_{S-1}[n]$.

$$\begin{aligned} d_{S-1}[n] &= d_S[n] - k_S \tilde{d}_{S-1}[n-1] \\ \tilde{d}_S[n] &= \tilde{d}_{S-1}[n-1] + k_S d_{S-1}[n] \end{aligned} \qquad (2.28)$$

Cette configuration permet donc d'effectuer un calcul de filtrage All-Pole d'ordre indéfini avec un vecteur d'UPMs. En outre la sortie $\tilde{d}_S[n]$ dépend de la valeur présente $d_{S-1}[n]$ et passée

$\tilde{d}_{s-1}[n\text{-}1]$. La figure 2-19 représente la cellule reconfigurée en filtre en treillis All-Pole au s^{ème} étage. Comme représenté, le module de récursivité est utilisé pour enregistrer la valeur passée de $\tilde{d}_{s-1}[n]$ qui est $\tilde{d}_{s-1}[n\text{-}1]$.

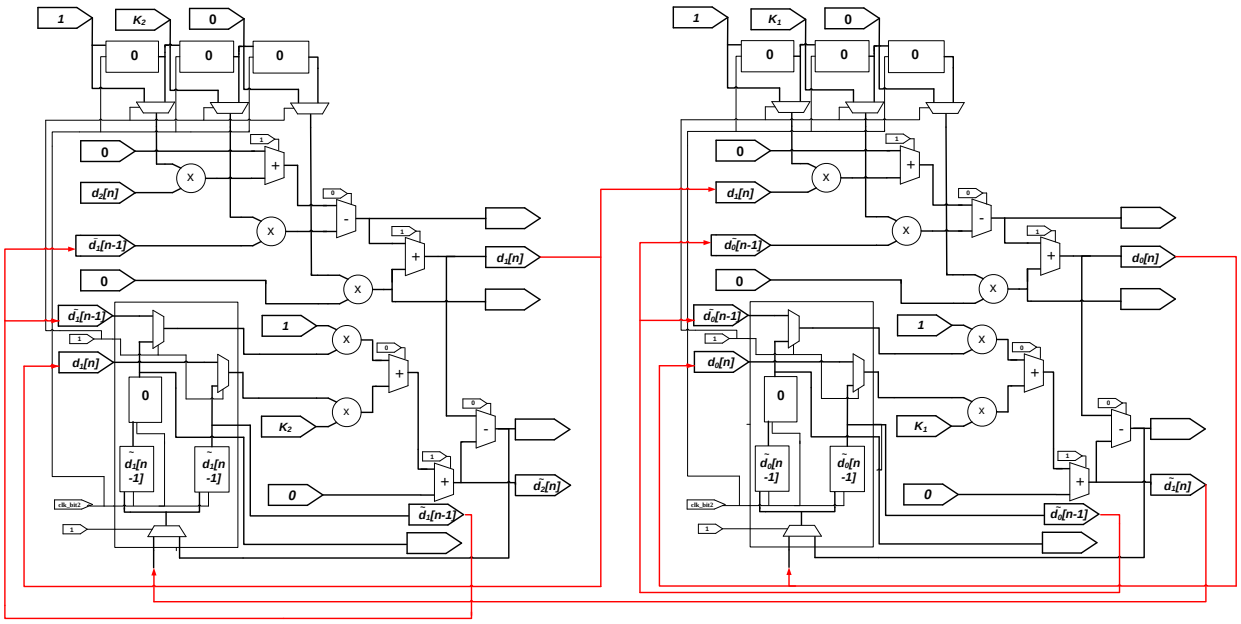


Figure 2-19 : Vecteurs d'UPM configurée en Filtre en Treillis All-Pole, *s* étages.

Les équations (2.4) à (2.10) peuvent être réécrites pour représenter un UPM configuré en filtrage All-Pole (Fig. 2-19) à l'aide des équations (2.29) à (2.35), en posant que la valeur de l'entrée de données *InX2* du premier UPM du vecteur égale à la valeur de l'entrée du processeur matriciel "*Samples*". Les valeurs des entrées de données *InA0*, *InX0*, *InD2* et *InX5* sont mises égales à *0*. Tous les UPMs du vecteur excepté le premier UPM ont une valeur de *InX2* qui est égale à la sortie *Out1* de l'UPM précédent. Pour toutes les cellules, la valeur de *bit1* est mise égale à *1* en binaire et la valeur de *bit3* est posée égale à *1* en binaire. Les valeurs respectives de *ad0*, *ad1*, *ad2*, *ad3*, *ad4*, *ad5* sont mises égales à *1*, 0, *1*, *1*, *1*, *0* en binaire. La sortie *Out5* de chaque UPM est connectée aux entrées *InX1* et *InX4* du même UPM. Pour chaque UPM la sortie *Out1* est connectée à *InX3* et la sortie *Out1* de chaque UPM (excepté le premier UPM du vecteur) est connectée à l'entrée *InX6* du même UPM :

$$Out0 = Samples * InX2 - InA1 * Out5 \qquad (2.29)$$

$$Out1 = Out0 + Out2 \tag{2.30}$$

$$Out2 = 0 \tag{2.31}$$

$$Out3 = Out1 - Out4 \tag{2.32}$$

$$Out4 = Out5 * InA4 \quad + Out1 * InA3 \tag{2.33}$$

$$Out5 = Out4 * Z^{-1} \tag{2.34}$$

$$Out6 = Out4 * Z^{-2} \tag{2.35}$$

2.5.4 Le Filtre Pole-Zero

Le circuit précédent est utilisé pour construire un filtre Pole-Zero. En effet le filtre Pole-Zero est la somme pondérée des résultats du filtrage All-Pole avec un gain C_S qui varie à chaque étage, [1] p775. L'équation (2.36) du filtre Pole-Zero est schématisée par la figure 2-20. La sortie $\tilde{d}_s[n]$ du filtre All-Pole est multipliée à chaque étage par une constante C_S. La sortie du pôles-zéros, $y_S[n]$, est le résultat de $\tilde{d}_s[n]$ multipliée par le paramètre C_S.

$$y_S[n] = \sum_{s=0}^{k} c_S \tilde{d}_S[n] \tag{2.36}$$

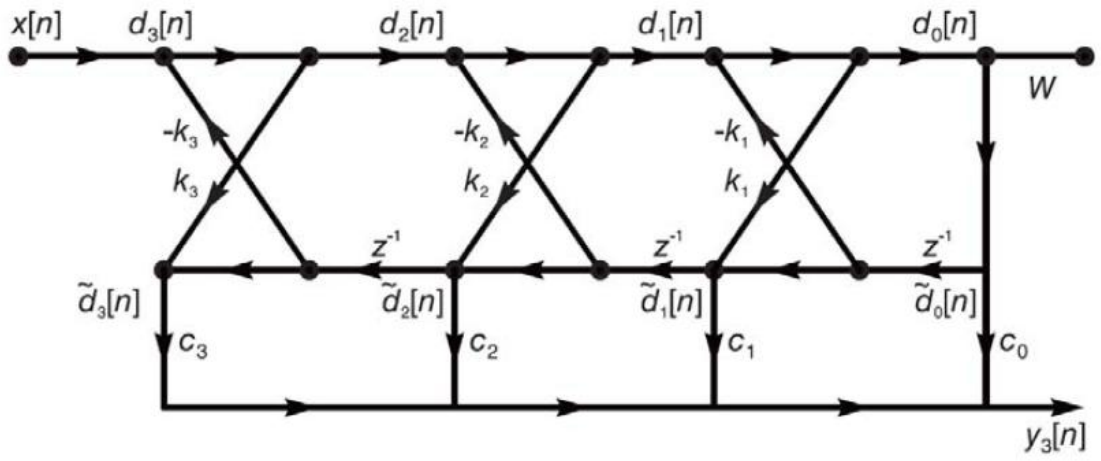


Figure 2-20 : Filtres en Treillis Pole-Zero[5] [1] p775.

2.6 Corrélation et Intercorrelation

La fonction de corrélation est décrite par l'équation suivante ou *v*[n] et *x*[n] sont les séquences d'entrées [1] p357-360:

$$r_{vx}[n] = \sum_{m=-\infty}^{\infty} v[n+m]x[m] \tag{2.37}$$

Pour un ordre égal à *3* on obtient les séquences suivantes :

$$r_{vx}[0] = v[0]x[0] + v[1]x[1] + v[2]x[2]$$
$$r_{vx}[1] = v[1]x[0] + v[2]x[1] + v[3]x[2]$$
$$r_{vx}[2] = v[2]x[0] + v[3]x[1] + v[4]x[2]$$

Pour un ordre égal à *5*, on obtient la séquence 6 suivante:

$$r_{vx}[4] = v[4]x[0] + v[5]x[1] + v[6]x[2] + v[7]x[3] + v[8]x[4] + v[9]x[5]$$

La figure 2-21 montre la séquence *4*, $r_{vx}[0]$, d'une corrélation d'ordre *N=5*. Il est donc possible de mettre en cascade d'un nombre indéterminé d'UPMs afin d'augmenter indéfiniment l'ordre de la corrélation. De la même manière, la fonction d'intercorrelation est décrite par l'équation suivante :

$$r_{xy}[k] = \sum_{n=-\infty}^{\infty} x[n+k]y[n] \tag{2.38}$$

La configuration de la figure 2-21 peut être utilisée pour calculer une intercorrelation, en remplaçant $x[n]$ dans (2.37) par $y[n]$ dans (2.38) et $v[n]$ dans (2.37) par $x[n]$ dans (2.38).

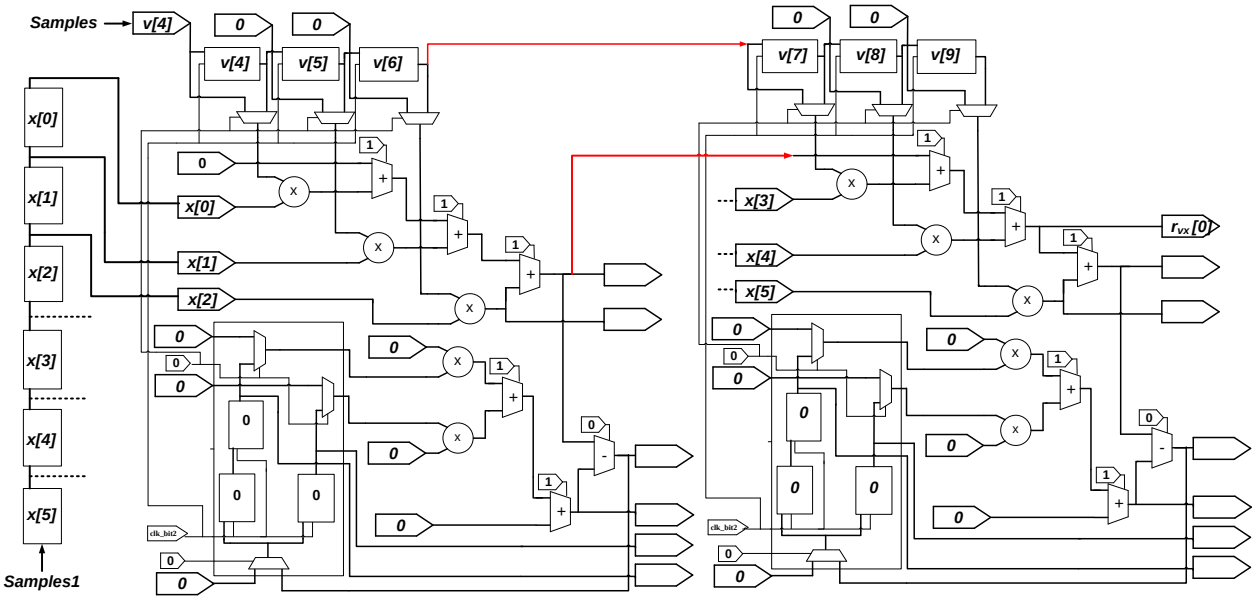


Figure 2-21 : Vecteur d'UPMs configurée en corrélation ordre N=5.

Les équations (2.4) à (2.10) peuvent être réécrites pour représenter un UPM configuré en filtrage FIR et corrélateur (Fig. 2-21) à l'aide des équations (2.39) à (2.43), en posant que la valeur de l'entrée de données *InA2* de l'UPM égale à la valeur de l'entrée du processeur matriciel "*Samples*"; en posant la valeur de *bit1* égale à *0* en binaire; la valeur de *bit3* égale à *1* en binaire; en posant les valeurs respectives de *ad0*, *ad1*, *ad2*, *ad3*, *ad4*, *ad5* égales à *1*, *1*, *1*, *1*, *1* , *0* en binaire. Les valeurs des entrées de données *InA0*, *InA1*, *InX5*, *InX3*, *InX4*, *InD2*, *InA4* et *InA3* sont mises égales à *0* en binaire :

$$Out0 = InA2 * Z^{-1} * InX2 + InA2 * Z^{-2} * InX1 \tag{2.39}$$

$$Out1 = Out0 + Out2 \tag{2.40}$$

$$Out2 = InA2 * Z^{-3} * InX0 \tag{2.41}$$

$$Out3 = Out1 - Out4 \tag{2.42}$$

$$Out4 = Out6 = Out5 = 0 \tag{2.43}$$

2.7 La transformée de Hilbert

David Hilbert fut un mathématicien Allemand qui vécut entre le XIX^{ème} et le XX^{ème} siècle. La transformée de Hilbert, $y[n]$ d'un signal $x[n]$ s'écrit comme suit :

$$y[n] = x[n] * h[n] \tag{2.44}$$

avec :

$$h[n] = \begin{cases} 0, \textit{ pour n pair} \\ 2/\pi n, \textit{ pour n impair} \end{cases}$$

Une transformée de Hilbert est équivalente à un filtre de réponse en fréquence décrite par l'équation suivante:

$$h(j\omega) = \xi[k] = \begin{cases} -j, \textit{ pour } \omega \textit{ positif} \\ j, \textit{ pour } \omega \textit{ negatif} \end{cases} \tag{2.45}$$

Ce qui correspond à un décalage de phase de *90* degrés. Si on calcule les valeurs de $h[n]$ pour $N=6$, on obtient :

$$h[0] = 0,\ h[1] = \frac{2}{\pi},\ h[2] = 0,\ h[3] = \frac{2}{3\pi},\ h[4] = \frac{1}{2\pi} \text{ et } h[5] = \frac{2}{5\pi}.$$

Le module de la figure 2-22 est configuré pour effectuer des transformées de Hilbert $N=6$. Afin d'augmenter l'ordre de l'opération, plusieurs de ces filtres peuvent être mis en cascade. Les équations (2.39) à (2.43) sont utilisées pour configurer les matrices d'UPMs en transformée de Hilbert.

2.8 La Transformée de Hartley Discrète

Proposée par R.V.L Hartley en 1942, la transformée de Hartley est étroitement liée à la transformée de fourrier. Cette opération à pour avantage de transformer les fonctions réelles en fonctions réelles, ([1], p936 et p938), et de même pour son inverse. La transformée discrète de Hartley a été introduite en 1983 par R.N. Bracewell. Soit $x[n]$ une séquence de n points, la transformée discrète de Hartley qui s'écrit $X_{Ha}[k]$ a pour équation :

$$X_{Ha}[k] = \sum_{n=0}^{N-1} \{\cos(kn2\pi/N) + \sin(kn2\pi/N)\} x[n] \tag{2.46}$$

Comme exprimée dans l'équation (2.46), cette opération représente la convolution des échantillons $x[n]$ avec la somme de cosinus et sinus de $2\pi kn/N$. La figure 2-23 illustre les différentes configurations de la transformée d'ordre *2*, pour la séquence $k=0$ et $n=0$. Le paramètre du filtre s'obtient comme suit :

$$b_{0,seq0} = \{\cos(0) + \sin(0)\} = 1$$

Les valeurs des paramètres de la séquence *1* sont calculées pour $n=0$ et $n=1$:

$$b_{0,seq1} = \{\cos(0) + \sin(0)\} = 1 \text{ et } b_{1,seq1} = \{\cos(\pi) + \sin(\pi)\} = -1.$$

La séquence *2* est écrite avec les coefficients calculés pour $n=0$, $n=1$ et $n=2$:

$$b_{0,seq2} = \{\cos(0) + \sin(0)\} = 1, b_{1,seq2} = \{\cos(2\pi) + \sin(2\pi)\} = 1, b_{2,seq2} = \{\cos(4\pi) + \sin(4\pi)\} = 1$$

Pour réaliser les autres configurations plusieurs UPMs peuvent être utilisés pour construire un vecteur de transformée de Hartley. Les équations arithmétiques (2.39) à (2.43) sont utilisées pour configurer les matrices d'UPMs en transformée de Hartley.

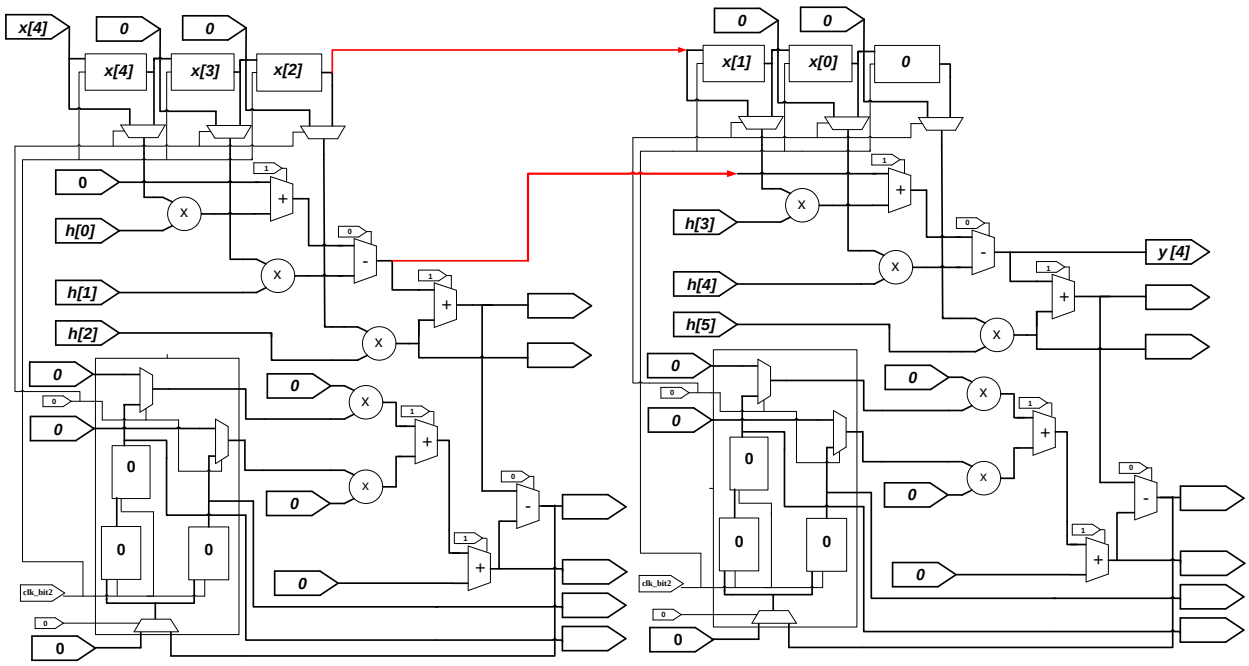


Figure 2-22 : Vecteur d'UPMs configurée en Transformée de Hilbert, ordre *N=6*, séquence *4*.

2.9 La Transformée de Cosinus Discrète (DCT)

La transformée de cosinus discrète se formule ainsi, [1] p946-948:

$$X_{DC}[k] = 2\sum_{k=0}^{N-1} x[n]\xi[k]\cos\frac{\pi}{N-1}nk \qquad (2.47)$$

avec :

$$k=0,1,\ \ldots,\ N\text{-}1 \ \text{ et } \ \xi[k] = \begin{cases} 1, \ pour\ n = 1,2,\ldots,N-2 \\ 1/2, \ pour\ n = 0,\ldots,N-1 \end{cases}$$

Pour un filtre d'ordre *N=2* et pour la *séquence k=0* :

$$\xi[0] = 1/2 \text{ et } \ 2\cos 0 = 2 \text{ donc } b_{0,seq0}=1.$$

Pour la *séquence k=1* :

$$n=0 : \ \xi[0] = 1 \ \text{ et } \ 2\cos 0 = 2 \text{ donc } b_{0,seq1}=2,$$

$$n=1 : \ \xi[0] = 1 \ \text{ et } \ 2\cos \pi = -2 \text{ donc } b_{1,seq1}=\text{-}2.$$

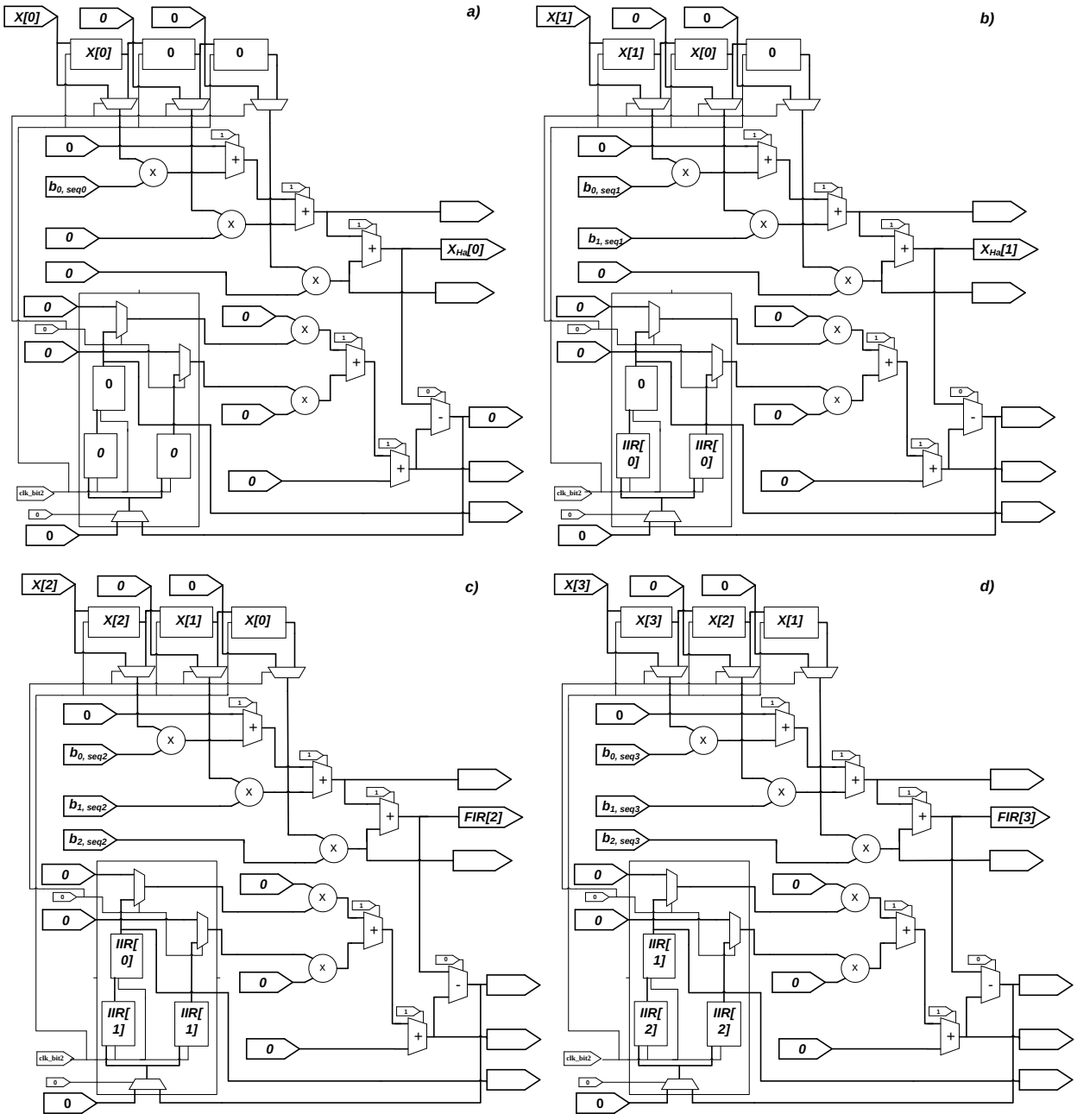


Figure 2-23 : Configuration en Transformée de Hartley *N*=2, *séquence 0* en a), *séquence 1* en b), *séquence 2* en c) et *3* en d).

Pour la séquence *k=2*,

$$n=0 : \xi[0] = 1/2 \text{ et } 2\cos 0 = 2 \text{ donc } b_{0,seq2}=1,$$

$$n=1 : \xi[0] = 1/2 \text{ et } 2\cos 2\pi = 2 \text{ donc } b_{1,seq2}=1,$$

$$n=2:\ \xi[0]=1/2 \text{ et } 2\cos 4\pi = 2 \text{ donc } b_{1,seq2}=1.$$

La figure 2-23 peut être utilisée pour calculer, respectivement, les *séquences 0*, *1*, *2* et *3*. Les équations arithmétiques (2.39) à (2.43) sont utilisées pour configurer les matrices d'UPMs en transformée de cosinus discrète.

2.10 La Transformée de Fourier Rapide (FFT)

La cellule logique proposée peut être employée comme des boites logiques pour construire des matrices de multiplieurs et d'additionneurs complexes en parallèle. Une matrice de multiplieurs et d'additionneurs complexes est capable de performer en temps réel une Transformée Discrète de Fourier DFT ou une Transformée Rapide de Fourier FFT d'une séquence donnée. Le même principe peut être employé pour construire une FFT avec un radical plus élevé, comme la FFT radical-*4* illustrée dans Fig. 2-24.

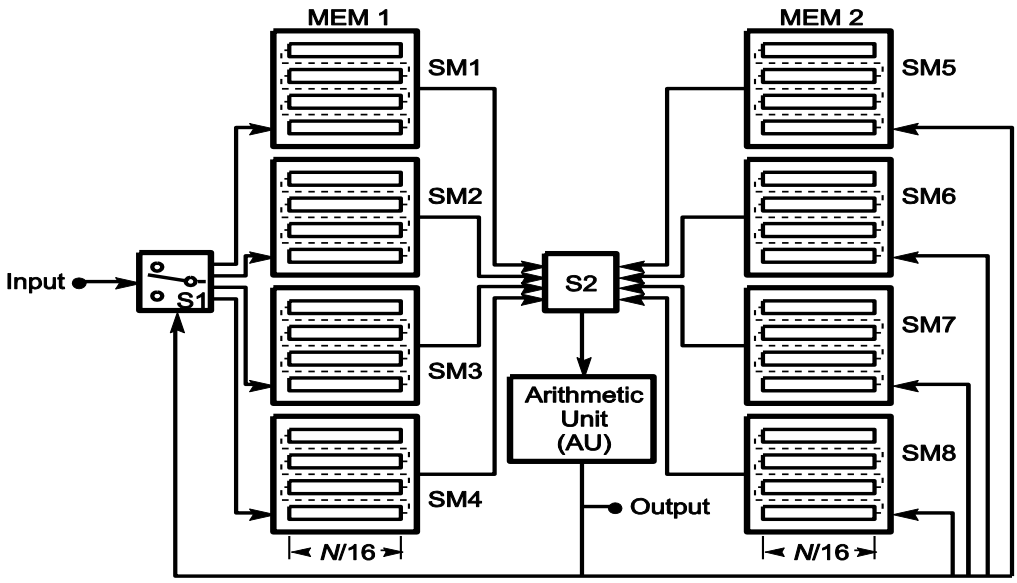


Figure 2-24 : Processeur FFT parallèle haute vitesse radical-*4*.

La figure 2-25 montre un processeur FFT algorithme OIOO, ordre *N*=4, (voir [1] p455-471, p1061-1062 et [10]), qui est utilisé pour construire un opérateur *butterfly* radical-4 [1] p458-459.

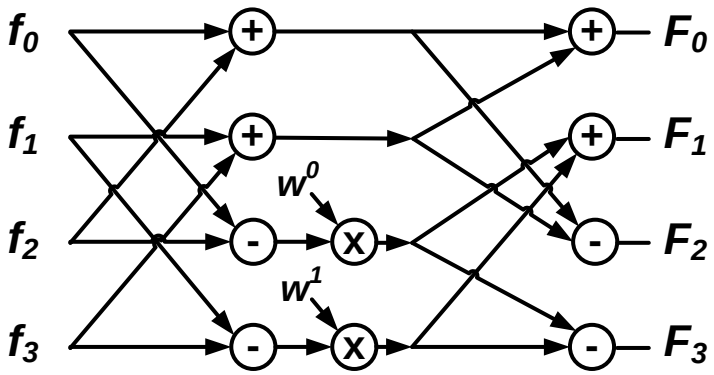


Figure 2-25 : Matrices d'UPM effectuant une FFT (OIOO) N=4.

En effet il est possible de construire des opérateurs "*Butterfly*" radical 2 et radical 4 avec respectivement deux et quatre UPMs. Le premier opérateur additionne et soustrait deux nombres complexes A et B et le résultat A-B est multiplié à un autre nombre complexe représentant $W=e^{-j2\pi/N}$ comme le montre la figure2-26 effectuant les opérations suivantes $A+B$, $A-B$ *et* $C=(A-B)W$.

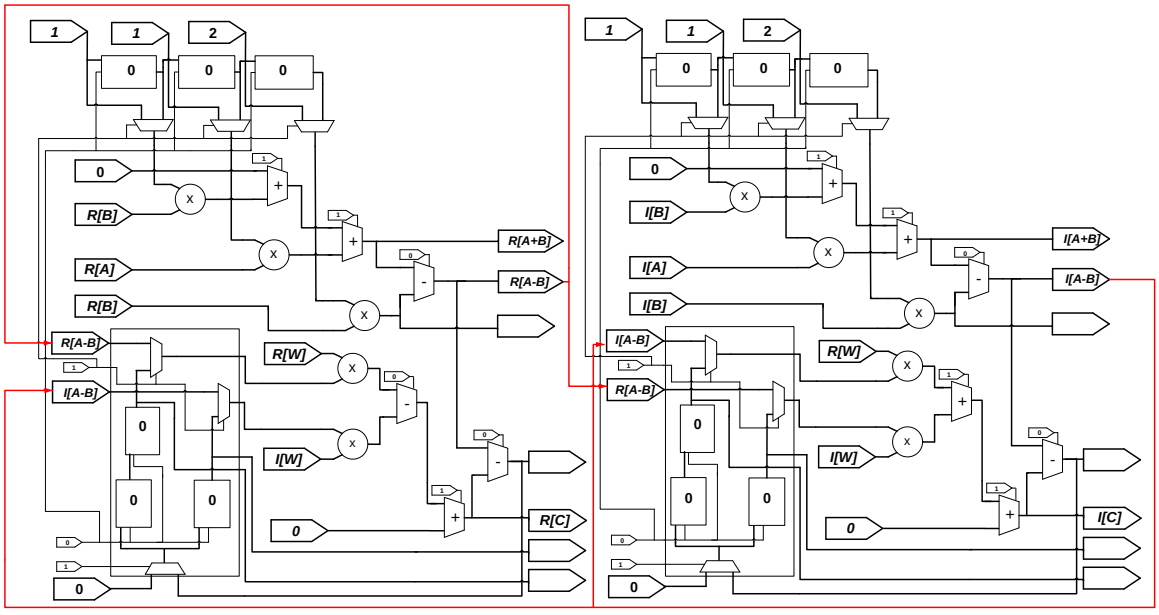


Figure 2-26 : Deux UPMs configurés en opérateur "*Butterfly*".

L'opérateur radical 4 est construit à l'aide d'une FFT (OIOO), ordre N=4 comme illustré dans la figure 2-27 a). Une matrice, de quatre opérateurs radical-4, est utilisée pour construire une FFT

N=16 comme illustré sur la figure 2-27 b). Les équations (2.4) à (2.10) sont simplifiées pour représenter un UPM configuré en opérateur *butterfly* radical-4 (Fig. 2-27) à l'aide des équations (2.48) à (2.53), en posant que la valeur de l'entrée de données *InA2* de l'UPM égale à la valeur de l'entrée du processeur matriciel "*Samples*". La valeur de *bit1* est mise égale à 1 en binaire et la valeur de *bit3* égale à *0* en binaire. Les valeurs respectives de *ad0, ad1, ad2, ad3, ad4, ad5* sont prises égales à *1, 1, 0, 1, 1* et *1* en binaire. La sortie *Out4* est connectée aux entrées *InX0* et *InX1*. Les valeurs des entrées de données *InX6, InD2* sont initialisées égales à *0*. Les deux UPMs du haut respectent les équations décrites cependant les configurations des deux UPMs du bas de la figure 2-27 sont les mêmes que ceux décrits par les équations (2.48) à (2.53) à la différence que la sortie *Out4* du dernier UPM et connectée aux entrées *InX0* et *InX1* de l'autre UPM et le bit de contrôle de l'adition *ad0* est négatif :

$$Out0 = InA2 * InX2 + InX5 + InA1 * Out4 \tag{2.48}$$

$$Out1 = Out0 - Out2 \tag{2.49}$$

$$Out2 = Out4 * InX0 \tag{2.50}$$

$$Out3 = Out1 + Out4 \tag{2.51}$$

$$Out4 = InX4 * InA4 \quad - InX3 * InA3 \tag{2.52}$$

$$Out5 = Out6 = 0 \tag{2.53}$$

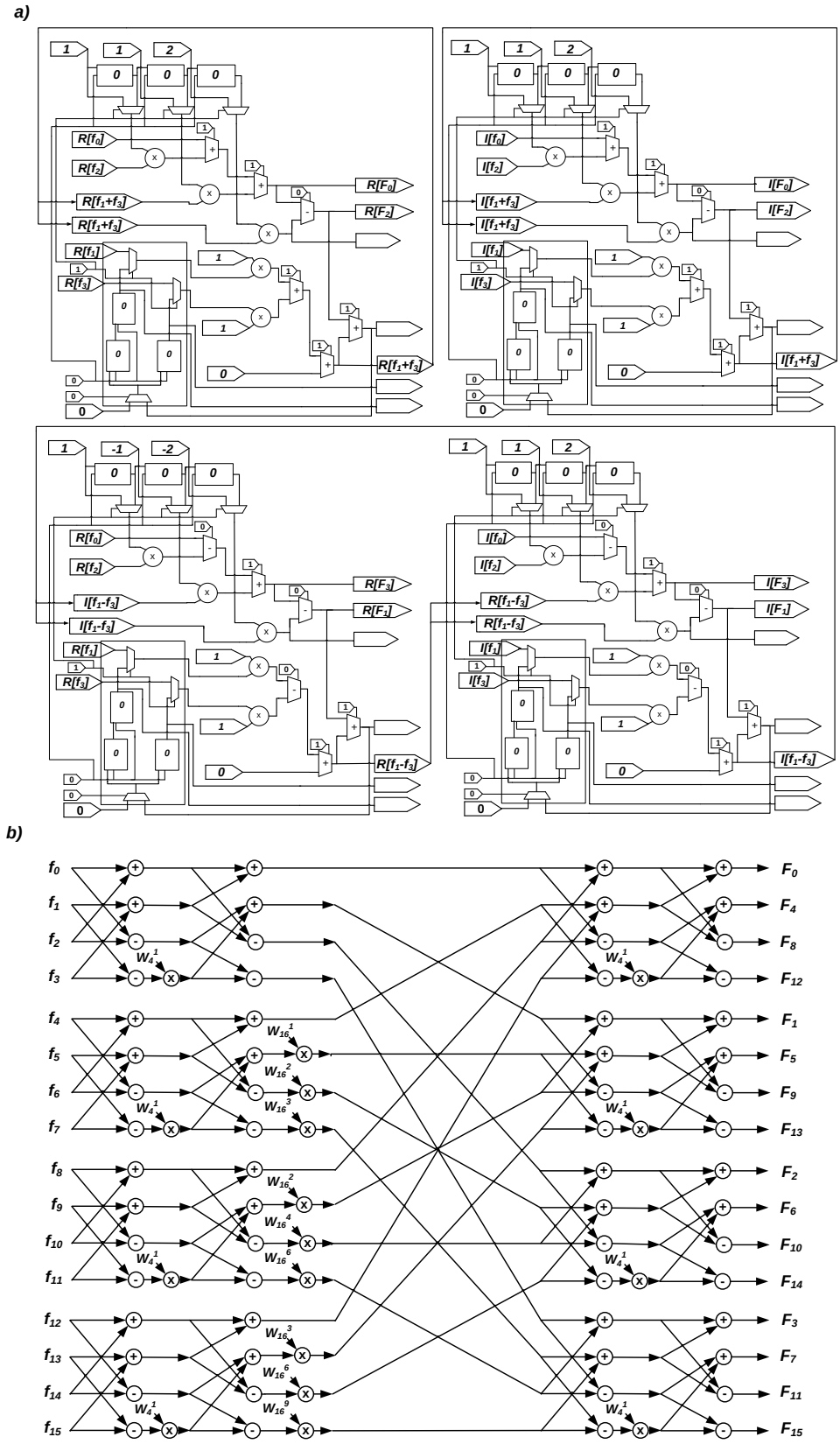


Figure 2-27 : Configuration en opérateur "*Butterfly*" radical 4 a) et FFT, ordre N=16.

2.11La Transformée de Walsh-Généralisée

Il est possible construire avec l'UPM, un processeur exécutant une transformée de Walsh-Hadamard, en ordre naturel, dyadique et séquencé. Plusieurs exemples de transformées de Walsh-Hadamard et de Walsh-généralisée qui sont proposés dans [29] et [1] p924-930 peuvent être prises comme références. Dans la figure 2-28, extraite de [1] p930, sont schématisées des transformées de Walsh-Paley et de Walsh-Kaczmarz pour N=27.

En effet l'architecture proposée dans [29,30] est optimale dans le sens qu'elles offrent un minimum de connexions, les données sont connectées aux entrées de manière à en avoir un accès rapide par décalage et l'optimisation au niveau du temps perdu est rendue maximale par le réarrangement des données. Les classes de matrices appelées "*General-Base Sampling Matrices*", les "*Span Matrices*" et "p^k*-Optimal Span Matrices*" associées sont appliquées aux algorithmes d'analyses spectrales généralisée telle que le "*Chrestenson Generalized Walsh*" aboutissant ainsi sur les transformées optimales de Walsh-Paley et de Walsh-Kaczmarz des signaux et images. La figure 2-29, illustre l'architecture optimale résultante d'un processeur parallèle base 5 p-optimal et base 5 p^2-optimal. De même qu'une transformation peut être appliquée pour éliminer les délais de réarrangement ou "*Shuffle*" sur la transformée de fourrier en deux dimensions obtenant ainsi un processeur d'image à géométrie constante parallèle la représentation graphique d'un processeur utilisant un UPE base 2 représenté par la figure 2-30, tirée de [10].

2.12 Génération de fonctions à l'aide de la série d'expansion de Chebyshev

L'approche proposée est utilisée pour la génération de fonctions utilisant les Séries d'expansion de Chebyshev comme présentée dans ce qui suit.

2.12.1 Génération de la fonction de puissance de *x*

Plusieurs configurations simples et optimales peuvent être utilisées pour générer des puissances de *x*. La figure 2-31, montre la génération de x^2, x^3 et x^4.

2.12.2 Génération des polynômes de Chebyshev

Les fonctions Trigonométriques, Exponentiel, Bessel, Gamma et d'autres peuvent être développées en séries de puissances utilisant les Polynômes de Chebyshev qui convergent plus rapidement que d'autres expansions telles que les séries de Taylor. Les polynômes de Chebyshev s'écrivent, voir [1] p1020-1027 :

$$T_n^*(x) = \cos(n\theta) \text{ avec } \theta = \cos^{-1}(2x-1) \tag{2.54}$$

Ces polynômes sont définis pour une expansion de fonction dans l'intervalle $0 \le x \le 1$. La Table 1 présente des polynômes pour $0 \le n \le 8$ et la figure 2-32 (a) montre la génération de T_1(x) et T_2(x) et la figure 2-32 (b) illustre la génération de T_4(x), [1] p1021.

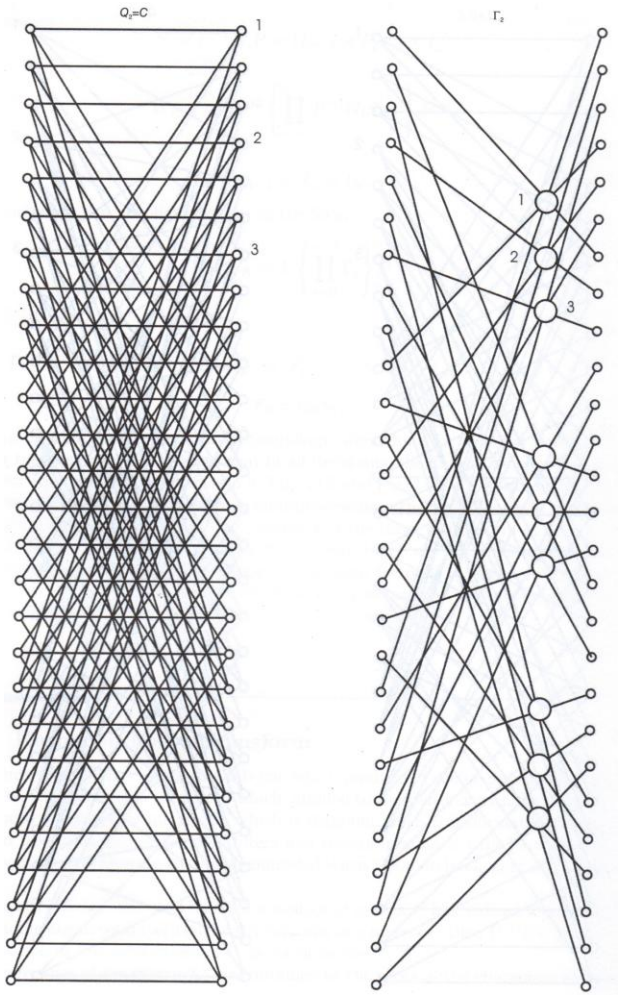


Figure 2-28 : Transformées de Walsh-Paley et de Walsh-Kaczmarz Généralisées [29].

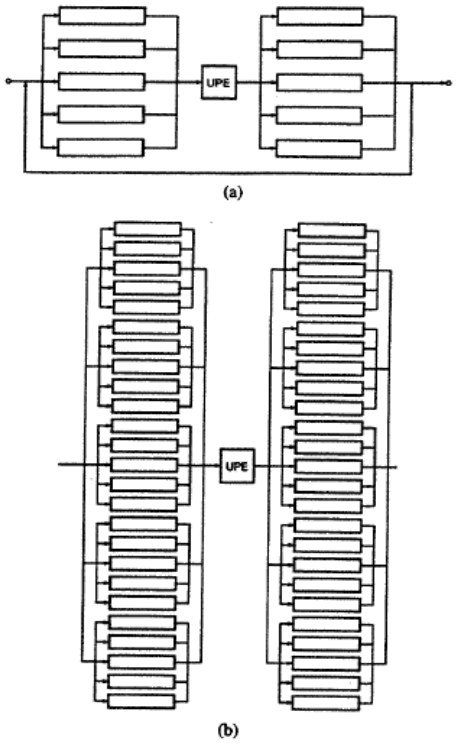


Figure 2-29 : Architecture optimale utilisant un UPE utilisé en a) base 5 p-optimal et b) les deux premiers étages d'un base 5 p^2-optimal avec pipeline [29].

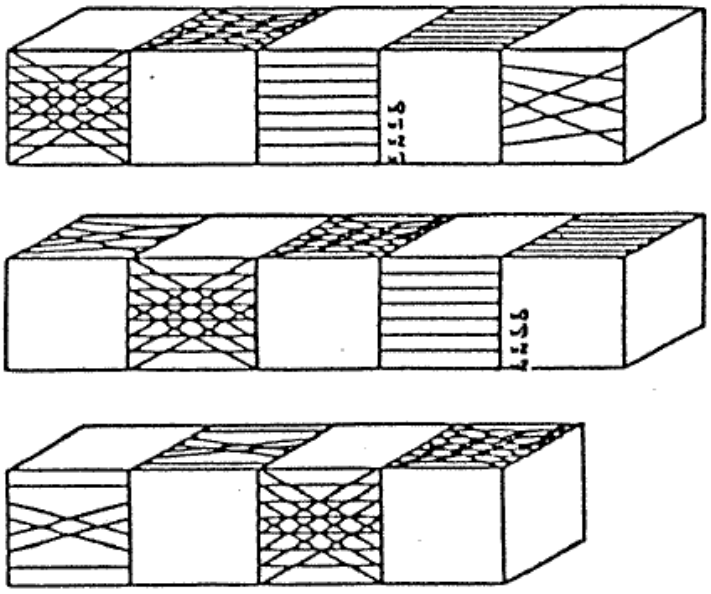


Figure 2-30 : Processeur d'image à géométrie constante parallèle utilisant un UPE base 2 [10].

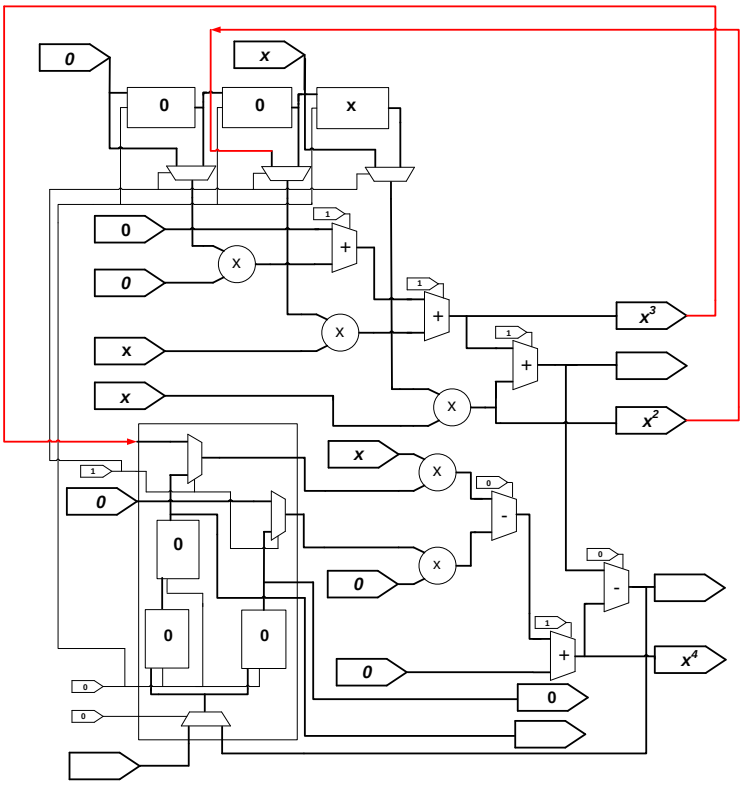


Figure 2-31 : Génération de x^2, x^3 et x^4.

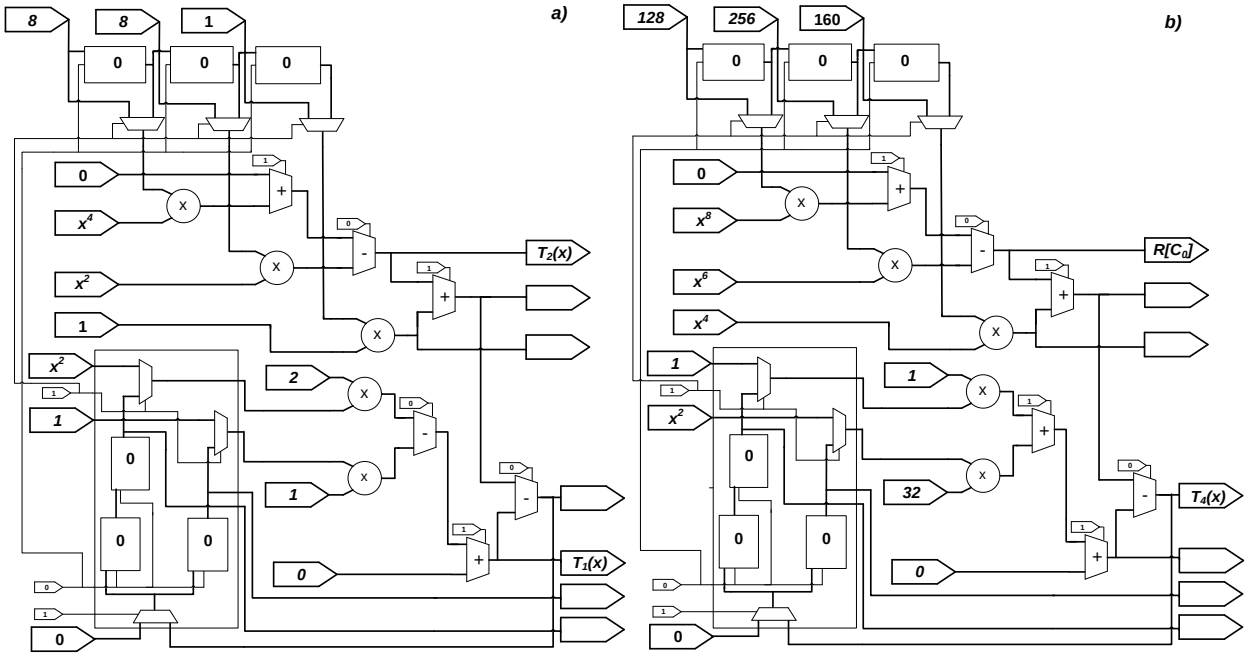


Figure 2-32 : Génération des polynômes de Chebychev T_1(x) et T_2(x) et T_4(x).

2.12.3 Génération de fonctions par les séries d'expansion de Chebyshev

L'approximation d'une fonction par les séries de puissances de Chebyshev est représentée par l'expression suivante [1] p1020-1023 :

$$f(x) = \sum_{n=1}^{\infty} a_n T_n^{*}\left(x^2\right) \tag{2.55}$$

Dans laquelle $T_n(x^2)$ représente les polynômes de Chebyshev tirés du tableau 1 et a_n sont les coefficients de Chebyshev.

Tableau 1 : Polynômes de Chebyshev pour $0 \le n \le 8$.

n	$T_n^*(x^2)$
0	1
1	$2x^2-1$
2	$8x^4-8x^2+1$
3	$32x^6-48x^418x^2-1$
4	$128x^8-256x^6+160x^4-32x^2+1$
5	$512x^{10}-1280x^8+1120x^6-400x^4+50x^2-1$
6	$2048x^{12}-6144x^{10}+6912x^8-3584x^6+840x^4-72x^2+1$
7	$8192x^{14}-28672x^{12}+39424x^{10}-26880x^8+9408x^6-1568x^4+98x-1$
8	$32768x^{16}-131072x^{14}+212992x^{12}-180224x^{10}+84480x^8+21504x^6+2688x^2-128x+1$

Le tableau 2 montre les coefficients de Chebyshev qui permettent de générer les fonctions trigonométriques et leur inverse, le tableau 3 présente les coefficients à employer pour générer des fonctions Exponentielles, Logarithmiques et Gamma, le tableau 4 montre les coefficients pour générer les fonctions Bessel. Par exemple la fonction cosinus peut être approximée en utilisant l'équation suivante :

$$\cos(\pi x/2) = \frac{a0}{2} + \sum_{n=1}^{\infty} a_n T_n^{*}\left(x^2\right) \text{ avec } -1 \le x \le 1 \tag{2.56}$$

Tableau 2 : Coefficient d'expansion de Chebyshev des fonctions Trigonométriques.

n	$sin(\pi x/2)$	$cos(\pi x/2)$	$Tan^{-1}(x)$	$sin^{-1}(x)$, $cos^{-1}(x)$
	a_n	a_n	a_n	a_n
0	2.552557925	0.9440024315	1.762747174	2.102463918
1	-0.2852615692	-0.4994032583	-0.1058929245	0.05494648722
2	0.009118016007	0.022799207962	0.01113584206	0.004080630393
3	-0.00013658751	-0.00059669519	-0.001381195	0.000407800685
4	$1.18496185*10^{-6}$	$6.70439487*10^{-6}$	0.00018574297	0.000046985367
5	$-6.7027916*10^{-9}$	$-4.6532296*10^{-8}$	$1.8574297*10^{-4}$	$5.88097581*10^{-6}$
6	$2.6672786*10^{-11}$	$2.1934576*10^{-10}$	$3.8210366*10^{-6}$	$7.77323124*10^{-7}$
7	$-7.8729221*10^{-14}$	$-7.4816487*10^{-13}$	$-5.6991862*10^{-7}$	$1.06774233*10^{-7}$

Tableau 3 : Coefficient d'expansion de Chebyshev des fonctions Exponentielles, Logarithmiques et Gamma.

n	e^x	e^{-x}	$Log(x+1)$	$\Gamma(x+1)$
	a_n	a_n	a_n	a_n
0	3.506775309	1.290070541	0.7529056258	1.883571196
1	0.8503916538	-0.3128416064	0.3431457505	0.00441581325
2	0.1052086936	0.03870411542	-0.02943725152	0.05685043682
3	0.008722104733	-0.00320868301	0.003367089256	-0.00421983539
4	0.000543436831	0.00019991924	-0.00043327588	0.00132680818
5	-0.00002711543	$-9.975211*10^{-6}$	0.0000594707119	-0.00018930245
6	$1.12813289*10^{-6}$	$4.15016897*10^{-7}$	$-8.50296754*10^{-6}$	0.00003606925
7	$4.02455823*10^{-8}$	$-1.4805522*10^{-8}$	$1.250467362*10^{-6}$	$-6.0567619*10^{-6}$

Tableau 4 : Coefficient d'expansion de Chebyshev des fonctions Bessel.

n	$J_0(x)$	$J_1(x)$
	a_n	a_n
0	0.06308122636	0.1388487046
1	-0.2146161828	-0.1155779057
2	0.004336620108	0.1216794099
3	-0.2662036537	-0.1148840465
4	0.3061255197	0.05779053307
5	-0.1363887697	-0.01692388016
6	0.03434754020	0.003235025204
7	-0.00569808232	-0.00043706086

La configuration de la figure 2-23 peut être utilisée pour calculer $\sum_{n=1}^{\infty} a_n T_n^*\left(x^2\right)$ en remplaçant les paramètres et les échantillons par polynômes de Chebyshev et des coefficients de Chebyshev. On peut ainsi utiliser la table 2 pour générer les fonctions trigonométriques (2.57) à (2.60) suivantes :

$$\sin(\pi x/2) = x\left(\frac{a0}{2} + \sum_{n=1}^{\infty} a_n T_n^*\left(x^2\right)\right) \text{ avec } -1 \le x \le 1 \tag{2.57}$$

$$\tan^{-1}(x) = x\left(\frac{a0}{2} + \sum_{n=1}^{\infty} a_n T_n^*\left(x^2\right)\right) \text{ avec -1≤x≤1} \tag{2.58}$$

$$\sin^{-1}(x) = x\left(\frac{a0}{2} + \sum_{n=1}^{\infty} a_n T_n^*\left(2x^2\right)\right) \text{avec } \frac{-1}{\sqrt{2}} \le x \le \frac{1}{\sqrt{2}} \tag{2.59}$$

$$\cos^{-1}(x) = \frac{\pi}{2} - x\left(\frac{a0}{2} + \sum_{n=1}^{\infty} a_n T_n^*\left(2x^2\right)\right) \text{ avec } 0 \le x \le \frac{1}{\sqrt{2}} \tag{2.60}$$

De la même manière le tableau 3 est utilisé pour générer les fonctions exponentielles, logarithmiques et Gamma (2.61) à (2.64) suivantes :

$$e^{x} = \frac{a0}{2} + \sum_{n=1}^{\infty} a_n T_n^*(x) \text{ avec 0≤x≤1} \tag{2.61}$$

$$e^{-x} = \frac{a0}{2} + \sum_{n=1}^{\infty} a_n T_n^*(x) \text{ avec 0≤x≤1} \tag{2.62}$$

$$\log(1+x) = \frac{a0}{2} + \sum_{n=1}^{\infty} a_n T_n^*(x) \text{ avec 0≤x≤1} \tag{2.63}$$

$$\Gamma(1+x) = \frac{a0}{2} + \sum_{n=1}^{\infty} a_n T_n^*(x) \text{ avec 0≤x≤1} \tag{2.64}$$

Le tableau 4 est utilisé pour générer les fonctions Bessel (2.65) à (2.66) suivantes :

$$J_0(x) = \frac{a0}{2} + \sum_{n=1}^{\infty} a_n T_n^*\left(x^2/100\right) \text{ avec -10≤x≤10} \tag{2.65}$$

$$J_1(x) = x\left(\frac{a0}{2} + \sum_{n=1}^{\infty} a_n T_n^*\left(x^2/100\right)\right) \text{ avec -10≤x≤10} \tag{2.66}$$

Une technique de calcul plus rapide et plus efficace consiste à reprendre l'équation (2.55) et y remplacer les polynômes de Chebyshev $T_n(x^2)$ par leurs propres valeurs en termes de x ([1] p1025). Par exemple dans l'équation (2.57), l'expression (2.67) a été développée en fonction de x, ce qui à pour effet de réduire considérablement le nombre de calculs.

$$\begin{aligned} f(x) &= \sum_{n=0}^{m} \alpha_n x^n = Sin(\pi x/2) = x\left(\frac{a0}{2} + \sum_{n=1}^{\infty} a_n T_n^{*}\left(x^2\right)\right) \\ &= 1.57079632x - 0.64596409x^3 \\ &+ 0.0796926261x^5 - 0.004681753389x^7 + 0.000160439053x^9 \\ &- 3.595706898.10^{-6}x^{11} + 5.462586570.10^{-8}x^{13} \end{aligned} \tag{2.67}$$

2.13 La Transformée Discrète de Hankel

La transformée discrète de Hankel est donnée par l'expression suivante[9] :

$$G(\rho) = 2\pi \sum_{r=0}^{\infty} rg[r] J_0[2\pi\rho r] \tag{2.68}$$

Dans laquelle $J_0[2\pi\rho r]$ est la fonction de Bessel générée par l'approximation de Chebyshev comme décrit aux sections précédentes. Pour un ordre N=2 et pour ρ=0, ρ=1 et ρ=2 on peut récrire l'expression (2.68) en fonction de r :

$$G[0] = 2\pi \sum_{r=0}^{2} rg[r] J_0[0]$$

$$G[1] = 2\pi \sum_{r=0}^{2} rg[r] J_0[2\pi r] = 0 * g[0] * J_0[0] + 2\pi * g[1] * J_0[2\pi] + 4\pi * g[2] * J_0[4\pi]$$

$$G[2] = 2\pi \sum_{r=0}^{2} rg[r] J_0[4\pi r]$$

La configuration de la figure 2-23 peut être utilisée pour le calcul de la transformée de Hankel. Les équations arithmétiques (2.39) à (2.43) sont utilisées pour configurer les matrices d'UPMs en transformée de Hankel et pour générer des polynômes de Chebyshev.

---

[9] Voir [1] p943-945.

2.14 La Division par convergence selon Newton-Raphson

La technique itérative de Newton-Raphson permet de calculer l'inverse d'un nombre dont le résultat multiplié à un dividende donne une division[10]. L'évaluation de l'inverse d'un nombre *B* peut se faire par l'équation récursive suivante de Newton-Raphson [1] p1016:

$$x_{i+1} = 2x_i - Bx_i^2 \tag{2.69}$$

Avec *x=1/B* et la condition initiale, $0 \le x_0 \le 2/B$. Un vecteur d'UPMs peut être construit afin d'évaluer l'inverse d'un nombre *B*, comme il est illustré dans la figure 2-33. Dans cette configuration la partie supérieure de chaque UPM est utilisée pour calculer une itération de l'équation récursive (2.69) en respectant les conditions initiales.

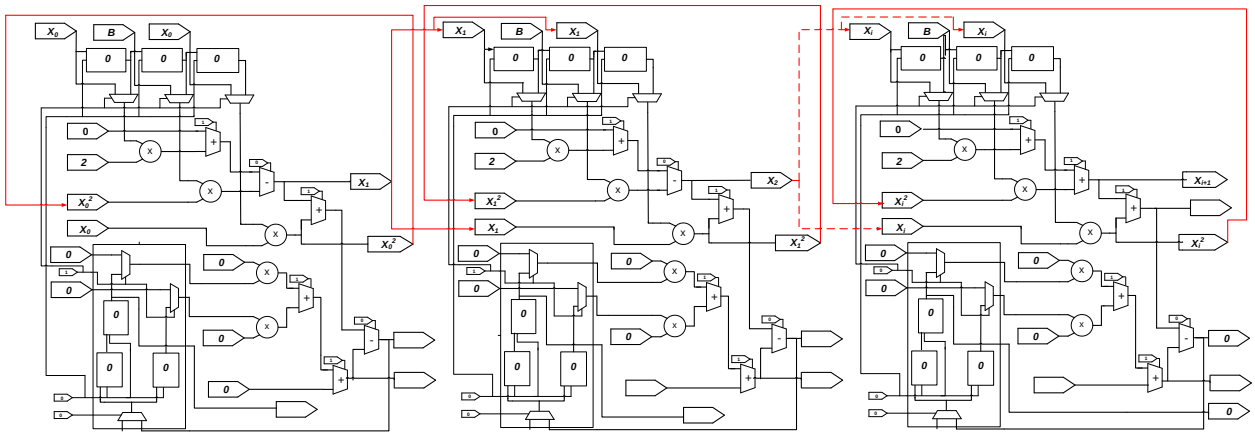


Figure 2-33 : Vecteur d'UPMs reconfigurés en calcul de division.

Les équations (2.4) à (2.10) peuvent être réécrites pour représenter un vecteur d'UPMs configuré en calcul de division (Fig. 2-33) à l'aide des équations (2.70) à (2.73), en posant que les valeurs des l'entrées de données *InX0*, *InA2* et *InA0* du premier UPM du vecteur sont égales à la valeur de l'entrée du processeur matriciel "*Samples*" et en posant que la valeur de l'entrée *InA1* de tous les UPMs égale à la valeur de *B*. La valeur de *bit1* est initialisée égale à 1 en binaire ainsi que la

[10] Voir [1] p1016-1017.

valeur de *bit3* est mise égale à *1* en binaire. Les valeurs respectives de *ad0*, *ad1*, *ad2*, *ad3*, *ad4*, *ad5* sont posées égales à *1*, *0*, *1*, *1*, *1* et *0* en binaire. La sortie *Out0* de chaque UPM est connectée aux entrées *InX0*, *InA0* et *InA2* de l'UPM suivant dans le vecteur. La valeur de l'entrée de données *InX1* est mise égale à *Out0* pour tous les UPMs, de même que les valeurs de *InA4*, *InA3*, *InX3*, *InX4*, *InD2* et *InX6* sont mises égales à 0.

$$Out0 = InA2 * InX2 - Out1 * InX1 \tag{2.70}$$

$$Out1 = Out0 + Out2 \tag{2.71}$$

$$Out2 = InA0 * InX0 \tag{2.72}$$

$$Out3 = Out4 = Out5 = Out6 = 0 \tag{2.73}$$

2.15 L'évaluation de la n^{eme} racine selon Newton-Raphson

De la même manière que dans la section précédente il est possible d'utiliser une matrice d'UPMs afin d'évaluer la n^{eme} racine d'un nombre *A*. En effet, l'équation récursive suivante permet de le faire en prenant $y = \sqrt[n]{A}$, [1] p 1018:

$$y_{i+1} = \left(\frac{n-1}{n}\right) y_i + \frac{A}{n y_i^{n-1}} \tag{2.74}$$

Par exemple, (2.74) est simplifiée pour *n=2* et est utilisée afin d'évaluer la racine carré d'un nombre A:

$$y_{i+1} = \frac{y_i}{2} + \frac{A}{2y_i} \tag{2.75}$$

Pour *n=3* :

$$y_{i+1} = \frac{2y_i}{3} + \frac{A}{3y_i^2} \tag{2.76}$$

Pour *n=4* :

$$y_{i+1} = \frac{3y_i}{4} + \frac{A}{4y_i^3} \tag{2.77}$$

Puisque l'équation (2.74) contient des divisions, la configuration du calcul de la division discutée précédemment est utilisée pour le calcul de la n^{eme} racine. En effet si on pose que $B = ny_i^{n-1}$ alors (2.74) peut s'écrire :

$$y_{i+1} = \left(\frac{n-1}{n}\right) y_i + \frac{A}{B} \tag{2.78}$$

Cette équation contient de deux divisions, qui peuvent être calculées par la méthode décrite à la section 2.14. Cette configuration d'UPMs est montrée dans la figure 2-34.

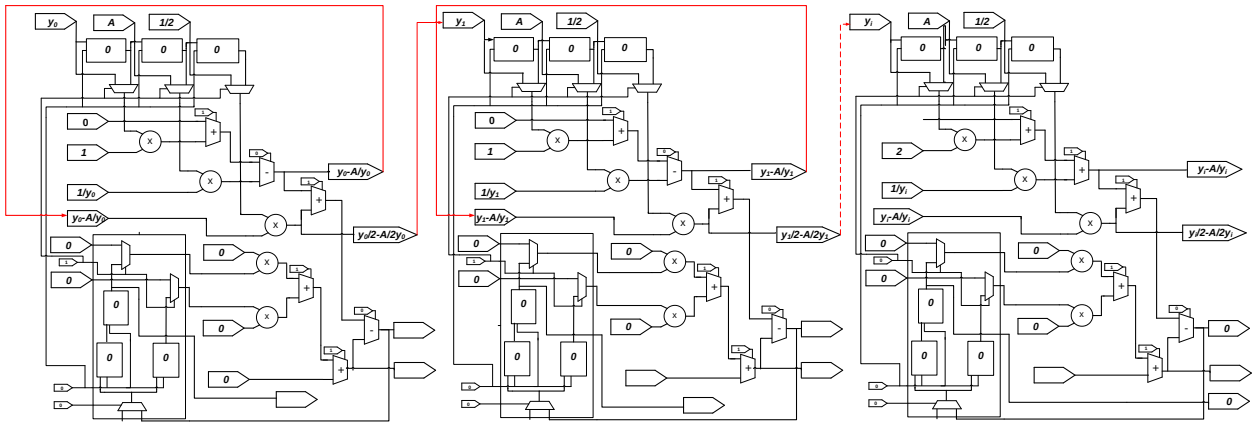


Figure 2-34 : Matrice d'UPMs reconfigurés en calcul de racine carrée.

Les équations (2.4) à (2.10) peuvent être réécrites pour représenter un vecteur d'UPMs configuré en calcul de racine carré (Fig. 2-34) à l'aide des équations (2.79) à (2.82), en posant que la valeur de l'entrée de données *InA2* du premier UPM du vecteur est égale à la valeur de l'entrée du processeur matriciel y_0. Les entrées de données *InA1* et *InA0* de chaque UPM du vecteur sont respectivement mises égales aux valeurs d'entrée du processeur matriciel *A* et ½. La valeur de l'entrée *InX0* de tous les UPMs est mise égale à la valeur de *Out0*. La valeur de *bit1* est mise égale à *1* en binaire ainsi que la valeur de *bit3* égale à *1* en binaire. Les valeurs respectives de *ad0, ad1, ad2, ad3, ad4, ad5* sont initialisées égales à *1, 0, 1, 1, 1* et *0* en binaire. La sortie *Out1* de chaque UPM est connectée à l'entrée *InA2* de l'UPM suivant dans le vecteur. La valeur de

l'entrée de données *InX0* est mise égale à *Out0* pour tous les UPMs et en posant les valeurs de *InA4*, *InA3*, *InX3*, *InX4*, *InD2*, *InX5* et *InX6* égales à 0.

$$Out0 = InA2 * InX2 - Out1 * InX1 \tag{2.79}$$

$$Out1 = Out0 + Out2 \tag{2.80}$$

$$Out2 = InA0 * Out0 \tag{2.81}$$

$$Out3 = Out4 = Out5 = Out6 = 0 \tag{2.82}$$

2.16 Conclusion

La conception théorique d'un module de traitement des signaux dédié aux matrices cellulaires vient d'être réalisée et l'originalité de ce module est qu'il peut effectuer avec les mêmes cellules la plupart des calculs de fonctions DSPincluant le filtrage récursif par l'intermédiaire de son module de récursivité, en plus des calculs proposés par les éléments de traitement discutés dans la section précédente. En effet ce processeur est aussi reconfigurable dans le sens qu'il utilise les mêmes cellules pour effectuer les fonctions proposées, c'est-à-dire que les mêmes matrices d'UPMs sont réutilisées pour effectuer des calculs de la plupart des fonctions de traitement de signaux.

CHAPITRE 3 CONCEPTION LOGICIELLE

Ce chapitre décrit dans un premier temps les outils de conceptions qui ont été utilisés pour la programmation et l'implémentation du Module de Traitement Universel sous forme d'un module de propriété intellectuelle (IP block). Les conceptions logicielles des sous-fonctions, du module de contrôle de récursivité, les conceptions de l'UPM et du programme test des sous-fonctions sur écran VGA sont présentées. Ce chapitre compte aussi parmi ces sections, la réalisation sur FPGA d'un processeur matriciel de 2x2 UPMs dont le contrôle des entrées et sorties sont expliqués ainsi que le contrôle temporel de la matrice et celui de la reconfiguration des connexions entre les cellules. Un autre processeur matriciel 6x4 UPMs a été conçu et testé en Vérilog-HDL pour diverses reconfigurations. Le logiciel Matlab *Simulink* a été utilisé pour programmer et simuler une matrice cellulaire 2x2. Ces matrices sont conçues de telles sortes qu'elles puissent être reconfigurées en plusieurs fonctions de traitement de signaux en utilisant les mêmes éléments de traitement. En outre ces cellules permettent d'effectuer des calculs de fonctions de traitement des signaux récursifs ou non récursifs avec un mot binaire à son entrée qui contrôle la reconfiguration des mêmes cellules.

3.1 Programmation de l'UPM et des sous-fonctions sur FPGA

3.1.1 Outils de conception

Un FPGA (Field Programmable Gate Arrays ou Réseaux de Portes Logiques Programmables) est un circuit intégré logique qui peut être configuré et reconfiguré au besoin par l'utilisateur[11], après sa fabrication, d'où le terme "Portes Programmables". Les fondateurs Ross Freeman et Bernard Vonderschmitt, de la société Xilinx, ont inventé cette technologie en 1985. Les FPGAs sont utilisés pour concevoir des processeurs numériques. Dans ce projet la carte de test FPGA "*Cyclone II FPGA Starter Developement Kit*" a été utilisé comme outil de conception. Cette carte de développement d'*Altera* comprend des fonctionnalités numériques qui permettent à

[11] Voir [1] p 1063

l'utilisateur de développer et de tester la conception d'un circuit dont la complexité va d'un simple circuit numérique à des projets multimédias complexes [47,48].

Le Verilog-HDL est un langage de description de matériel informatique utilisé pour décrire les circuits intégrés et les systèmes d'opérations. Il est l'un des deux HDL standards IEEE (*IEEE Standard 1364-2001*) utilisé avec le VHDL (*IEEE Standard 1076-2002*). Ce langage a été lancé, ainsi que son simulateur, en 1983 par la corporation Gateway Design Systems. Dans le langage Verilog-HDL on spécifie des modules qui décrivent la conception du circuit et qui contiennent toutes les fonctions et procédures internes au circuit. Un programme Verilog-HDL est constitué d'un ou plusieurs modules qui sont connectés entre eux. Chaque module est écrit sur un fichier indépendant dont l'extension du nom est ".v". Chacun de ces modules a un port d'interface qui décrit la manière dont ils sont interconnectés entre eux. Le module "*Top-Level*" est le premier module dans la hiérarchie et il contient tous les autres modules. Il spécifie comment les autres modules sont utilisés et comment sont définis les ports d'entrées/sorties du circuit. En outre le langage Verilog utilise les formats de données "*wire*" et "*reg*". Le type de données "*wire*" est équivalent à un fil électrique qui interconnecte les éléments du circuit et le type de données "*reg*" est une variable qui change en fonction de l'état de la procédure, c'est-à-dire qu'il enregistre la dernière valeur qui lui a été assignée.

Le projet réalisé dans ce mémoire a été conçu en utilisant les versions v.7.2, v.9.1 et v.10.1 du logiciel de développement "*Quartus II*" qui est un outil de développement logiciel, produit par *Altera*, pour l'analyse et la synthèse des circuits HDL, qui permet à l'utilisateur de compiler son programme, de faire une analyse temporelle de circuit logique, de produire des diagrammes RTLs et de simuler les circuits en réaction à un stimulus [49]. Une version gratuite de ce logiciel peut être utilisée pour tester l'intégralité de ce projet Verilog-HDL [50].

Dans une réalisation électronique un cœur de propriété intellectuelle ou "*Intellectuel Proprety Core*" est une unité logique, cellule ou puce qui peut être réutilisable et qui est la propriété intellectuelle d'une partie. Les "*IP cores*" peuvent être utilisés comme des blocks de conception dans les ASICs ou FPGAs [52].

3.1.2 Génération de sous-fonctions par l'environnement de programmation

Le logiciel *Quartus II* offre une librairie de fonctions logiques préconçues et " *IP cores* " appelée *Megafunction/LPM* qui comprend des unités de traitement arithmétiques, des portes, des entrées / sorties, des mémoires et des unités de stockage. Le multiplexeur de *32-bits*, utilisé dans cette réalisation, a été généré et appelé *Multiplexeur32b*.v par l'intermédiaire de cette librairie de fonction dans laquelle elle est appelée *LPM_MUX*. De même qu'un registre à décalage de 32bits appelé *LPM_SHIFTREG* a été généré et nommé *Registre32b*.v ainsi que des additionneurs et multiplieurs point flottant en précision simple, appelés respectivement *ALTFP_ADD_SUB* et *ALTFP_MULT*.

3.1.3 Construction du Module de Récursivité

Le programme du module de récursivité, appelé *ControleRecurs.v* est montré dans l'ANNEXE 2 avec les entrées et sorties respectant l'algorithme correspondant à la figure 3-1 comme décrit dans les sections théoriques précédentes. Ce code débute avec la déclaration du module *ControleRecurs*, des entrées et sorties et des connexions internes. Les fonctions *multiplexeur* et registre à décalage sont appelées suivant la logique de la figure 3-1. La figure 3-2 illustre le diagramme RTL du module de contrôle de récursivité et montre le nombre de bits pour chaque entrée et sortie. L'ANNEXE 3 montre le diagramme interne du module de récursivité comprenant les sous-fonctions, les entrées, les sorties et les connexions internes entre les sous-fonctions.

3.1.4 Construction de l'UPM

Le Module de Traitement Universel est construit de manière à ce qu'il puisse être reconfiguré dans le but d'effectuer les algorithmes de traitements décrits dans le chapitre précédent. Le programme appelé *UPM*.v est montré dans l'ANNEXE 4 et les entrées et sorties de ce module correspondent à celles illustrées sur la figure 3-3. Ce code Verilog-HDL consiste en un appel de la fonction module de récursivité suivie de plusieurs appels successifs des fonctions additionneurs et multiplieurs point flottant en respectant l'architecture de la figure 3-3. Dans l'ANNEXE 5 est illustré le diagramme interne de l'UPM avec ses sous-fonctions et connexions mutuelles.

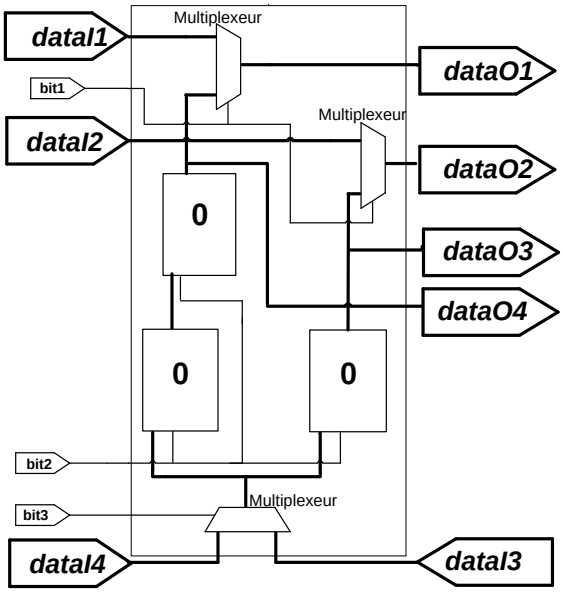


Figure 3-1 : Noms des entrées et sorties du Module de contrôle de la récursivité.

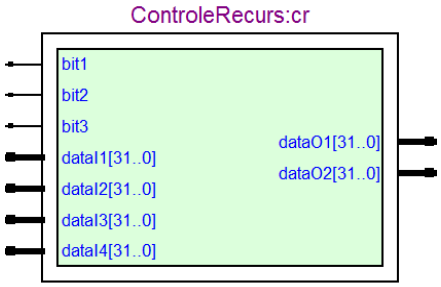


Figure 3-2 : Diagramme RTL du Module de contrôle de la récursivité.

3.1.5 Conception du programme test sur écran VGA

Le "*Cyclone II FPGA Starter Developement Kit*" est un kit de développement de circuit FPGA qui comprend la carte de développement, appelée le "*Cyclone II FPGA Starter Developement Board*" utilisée avec la version v.7.2 de "*Quatus II*". Le Module Universel de Traitement a été implémenté sur cette carte de développement par l'intermédiaire du logiciel "*Quartus II*" afin d'être testé. L'ANNEXE 6 montre une image de la connexion *USB-Blaster* entre la carte de développement et l'ordinateur contenant l'environnement de programmation et l'image de la connexion entre la carte et l'écran VGA.

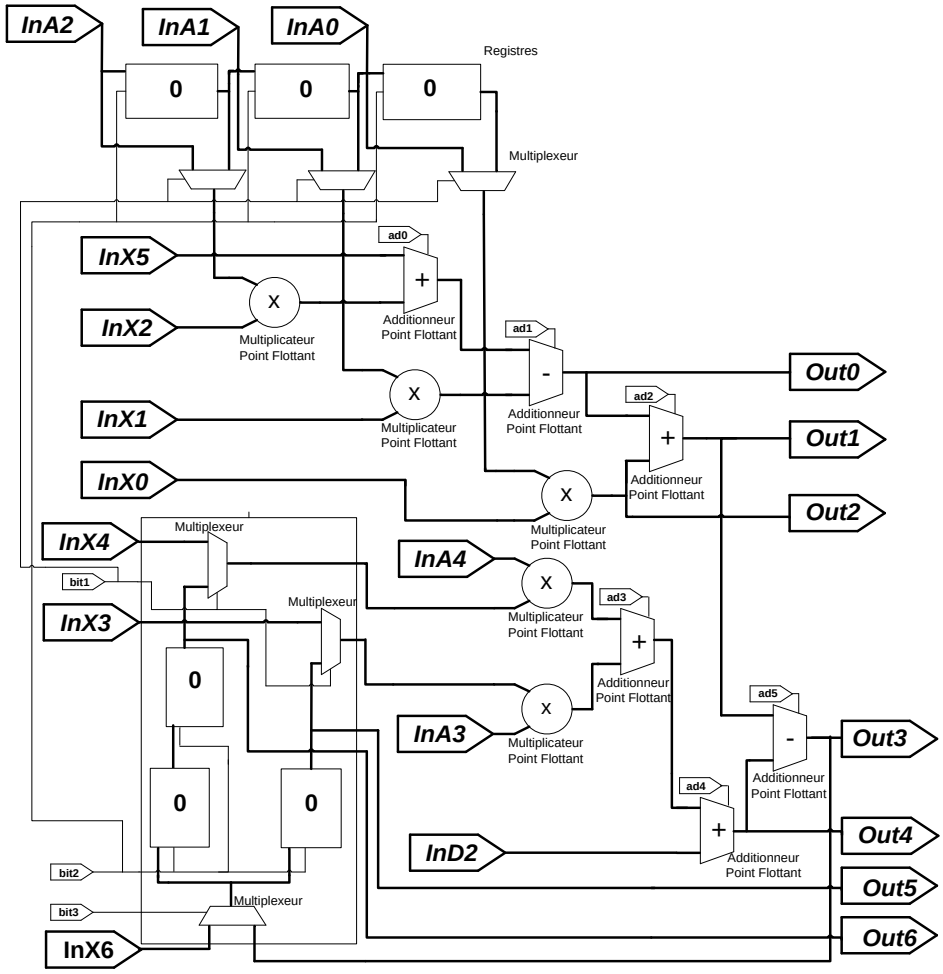


Figure 3-3 : Entrées et sorties de l'UPM.

Le "*Cyclone II FPGA Starter Developement Board*", contient aussi des programmes qui permettent d'utiliser les fonctions de la carte telle que l'affichage VGA [47]. En effet, ce kit fournit un programme test appelé "*CII_Starter_Default*" qui est modifié dans ce projet afin de réaliser les tests sur écran VGA. Une partie de ce programme Verilog-HDL, écrit par *Altera*, est contenue dans les ANNEXEs 7 et 8 qui montrent le code du module-top "*CII_Starter_Default*", qui est le module le plus élevé dans la hiérarchie des fonctions de ce programme. Dans ce programme, sont déclarées les entrées et sorties de la carte de développement telles que les horloges de *24, 27* et *50-MHz* des boutons poussoirs "*Push Button*", des interrupteurs "*DPDT Switch*", un protocole de communication "*I2C*", un protocole de communication "*USBJTAG*", des entrées et sorties de contrôle VGA et *40* pins d'entrées sorties "*GPIO*". Cette fonction aussi

programmée dans l'ANNEXE 8 (2) appelle d'autres modules de hiérarchies inférieures nommés "*VGA_Audio_PLL*", "*VGA_Controller*" et "*VGA_Pattern*". La fonction "*VGA_Audio_PLL*" est une boucle à verrouillage de phase ou "*Phase-Locked Loop*" qui permet d'asservir l'horloge d'entrée de *27-MHz* aux horloges audio et VGA qui correspondent aux sorties du module PLL. La fonction "*VGA_Controller*" est un programme de contrôle d'affichage VGA qui permet de contrôler la sortie VGA de la carte. Ces deux dernières fonctions sont écrites par *Altera* et ne seront pas détaillées dans ce mémoire pour des questions de concisions. Le programme "*VGA_Pattern*" est un code qui permet de décrire le contenu de l'affichage qui sortira sur l'écran VGA. La figure 3-4 présente le diagramme interne de la fonction "*CII_Starter_Default*" qui contient les fonctions internes, "*VGA_Audio_PLL*", "*VGA_Controller*" et "*VGA_Pattern*".

Le programme "*VGA_Pattern"*, en ANNEXE 9, 10 et 11 ne représente qu'une petite partie du code utilisé pour l'affichage de ce test et il est destiné à donner au lecteur de ce mémoire une idée sur comment utiliser un FPGA pour afficher des valeurs sur un écran VGA. En effet le code des tests d'affichage sur écran VGA ne sera pas inclus dans ce mémoire pour des questions de concisions. L'ANNEXE 12 montre une image de l'affichage des tests sur un écran VGA.

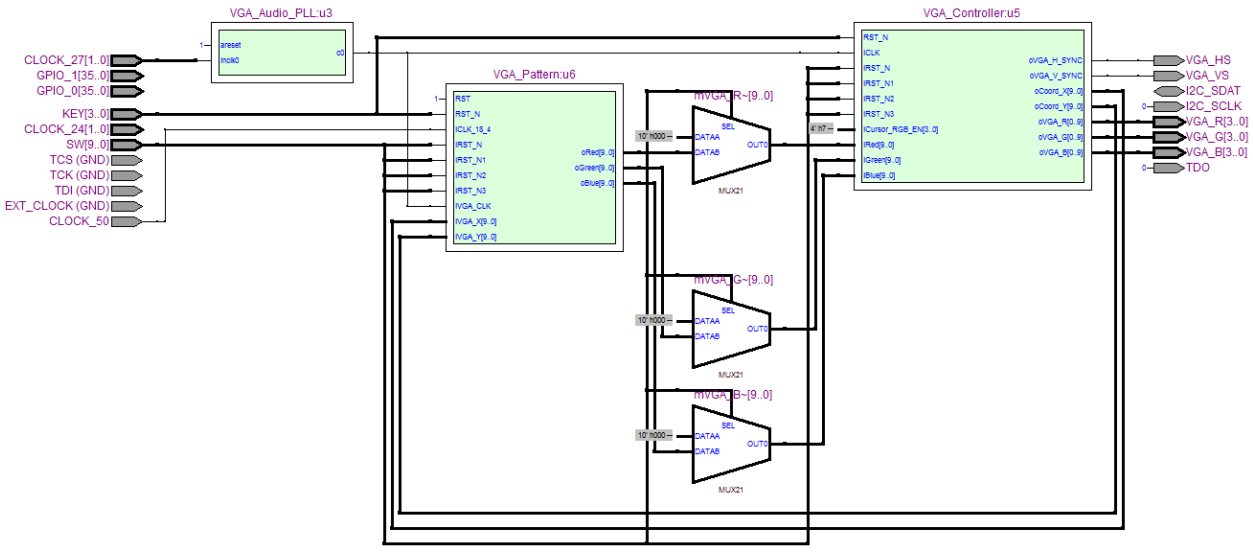


Figure 3-4 : Diagramme interne du module "*CII_Starter_Default*".

Le code de "*VGA_Pattern*" décrit comment les entrées et sorties sont déclarées, comment sont appelées les fonctions et comment les affichages des lettres "E" de "ECOLE" et U de "UPM" sont programmées sur l'image de l'ANNEXE 12. Le code de "*VGA_Pattern*", montré dans les ANNEXEs 9, 10 et 11, reçoit les entrées et sorties du contrôleur VGA, une horloge "*iCLK_18_4*", un bouton poussoir "*RST_N*", des interrupteurs nommés "*iRST_N*","*iRST_N1*" et "*iRST_N3*". La carte de développement "*Cyclone II FPGA Starter Developement Board*" offre quatre boutons poussoirs et dix interrupteurs qui permettent de modifier le contenu de l'affichage en fonction de l'état "*high*" ou "*low*" de ces boutons. En effet leurs valeurs sont contenues dans les entrées des programmes "*CII_Starter_Default*" et "*VGA_Pattern*" nommées "*Key[0]*" et "*SW[0]*". Ce code débute, en ANNEXE 9 avec une déclaration des constantes qui vont servir d'entrées aux fonctions à tester. Puis les entrées et sorties du contrôleur VGA sont déclarées ainsi que les entrées de contrôle de *1-bit*, les sorties des fonctions à tester et des registres internes. Le programme appelle ensuite les fonctions addition, soustraction et multiplication point flottant puis la fonction UPM configurée en additionneur et multiplieur complexe, fonction "*Butterfly*" et FFT, ordre *N*=4. Les sorties de ces fonctions, représentants les résultats des calculs, ont été déclarées "*wire*" et sont connectées à la sortie VGA par l'intermédiaire du code écrit en ANNEXE 10 et 11. En effet dans l'ANNEXE 10, à chaque montée de l'horloge VGA ou si le bouton poussoir "*RST_N*" est à "*low*" et si l'interrupteur "*iRST_N2*" est à "*low*" alors l'écran affiche le code qui suit. L'écran VGA est divisé en deux axes de *x=640* pixels horizontaux et *y=480* pixels verticaux. Ci-dessous, est programmé un exemple d'affichage d'une barre horizontale, de couleur rouge d'intensité *450*, qui va de la coordonnée *y=10* à *y=50* sur l'axe horizontal et de *x=405* à *x=408* sur l'axe vertical.

```
begin
oBlue   <=   (iVGA_Y<10)                                   ?   1000:
                                                                1000;
             ////////////////////////////////////////////////////////////////////////////////
oRed    <=   (iVGA_Y<010)                                                          ?   0:
             //Ecriture d'une barre verticale en rouge d'intensité 450
             (iVGA_Y>=10 && iVGA_Y<50 &&iVGA_X>=405 && iVGA_X<418)                  ?   450:
                                                                                        0;
             ////////////////////////////////////////////////////////////////////////////////
oGreen  <=   (iVGA_Y<10)                                                        ?   0:
                                                                                    0;

end
```

De cette manière il est possible d'afficher des lettres comme le montre l'ANNEXE 12. Les ANNEXES 10 et 11 présentent un exemple d'affichage des lettres "E" de "ECOLE" et U de

"UPM". Il est possible de mixer les couleurs en modifiant le même code d'affichage dans les sections dédiées à la couleur bleue, rouge et vert en changeant leurs intensités. Dans le code de l'ANNEXE 11 figure aussi l'affichage des lettres "*I*" et "*d*" de "*Id*" qui correspond à la dernière case en bas à droite dans l'espace graphique de l'image de l'ANNEXE 12. Ce nom correspond à la partie imaginaire du résultat *d* de la FFT, ordre *N*=4. L'affichage de valeur du résultat *d* de à la partie imaginaire de la FFT, ordre *N*=4 est aussi programmée dans l'ANNEXE 12, par l'intermédiaire de *32* petites dents représentants les *32-bits* de la mantisse, de l'exposant et du signe du résultat point flottant de la partie imaginaire de *d*. Les bits qui sont à *1* sont plus élevés que les bits qui sont à *0*. Le code suivant montre l'affichage des bits *28*, *27* et *26* du résultat "*result_Id_FFT*" sur la même ligne *y=350* :

```
 iVGA_Y>=(350-result_Id_FFT[28]) && iVGA_Y<352 &&iVGA_X>=545 && iVGA_X<546)  ?          750:
(iVGA_Y>=(350-result_Id_FFT[27]) && iVGA_Y<352 &&iVGA_X>=547 && iVGA_X<548)  ?          750:
(iVGA_Y>=(350-result_Id_FFT[26]) && iVGA_Y<352 &&iVGA_X>=549 && iVGA_X<550)  ?          750:
```

Si le bit *28* de "*result_Id_FFT*" est à *1* alors le code affichera aux coordonnées (*545, 352*) une dent de largeur *1* pixel et de hauteur *2* pixels. Si le bit *28* de "*result_Id_FFT*" est à *0* alors la dent sera de hauteur *1* pixel. C'est de cette manière que tous les bits de tous les résultats des appels de fonctions sont affichés dans ce test.

3.2 Conception de matrices d'UPMs reconfigurables sur FPGA

Une matrice de 2x2 UPMs est programmée dans cette partie en Verilog-HDL et son code est programmé dans les ANNEXES 13 à 18. Cette matrice est simulée pour différentes reconfigurations tel que le filtrage FIR/IIR, ordre *N*=9, le filtrage All-Pole, *s*=4 étages, le filtrage All-Zero *s*=4 étages qui peuvent être utilisées de manière adaptive ou non adaptive. Le programme contient aussi un opérateur "*Butterfly*" FFT radical-4, un algorithme de division selon Newton-Raphson *s*=4 étages. L'acheminement des données, des ports d'entrées vers cette matrice 2x2, est présenté ainsi que la sortie des résultats des cellules vers les ports de sorties du processeur. Le contrôle temporel de la matrice 2x2 est expliqué dans les sections qui suivent ainsi que la reconfiguration des connexions entre les cellules pour chaque mode de fonctionnement. Par la suite une seconde matrice de 6x4 est conçue en Verilog-HDL dans les ANNEXES 19 à 38 afin de tester les mêmes et d'autres reconfigurations avec un plus grand nombre de cellules. Les filtrages adaptatifs FIR/IIR, ordre *N*=20 et FIR d'ordre *N*=28, sont réalisés ainsi qu'une FFT

N=16 radical-4 et un algorithme de calcul de la racine carrée selon l'algorithme de Newton-Raphson *s*=4 étages qui utilise, pour chaque étage de calcul de racine, l'algorithme de division selon Newton-Raphson *s*=3 étages. Ce processeur a aussi été réalisé et simulé sur Matlab *Simulink* pour les fonctions FIR/IIR, ordre *N*=9 et FIR d'ordre *N*=12, ainsi qu'une FFT radical-4, un algorithme de calcul de la Racine Carrée selon l'algorithme de Newton-Raphson *s*=1 et l'algorithme de division selon Newton-Raphson *s*=4 étages.

3.2.1 Configuration des ports d'entrées et sorties du processeur matriciel 2x2

Dans cette section les ports d'entrées et sorties du processeur matriciel 2x2 sont présentés de même que la manière dont les données, provenant de ces entrées périphériques, sont acheminées vers les cellules. Cette partie explique aussi comment les résultats des calculs effectués par les UPMs, sont envoyés vers les ports de sorties du processeur. Cette matrice reçoit les données à partir de deux entrées de *32-bit* nommées respectivement, *Samples* et *Samples1*. L'entrée appelée *FP_clock* est dédiée à une horloge de *1-bit* utilisée par les operateurs point flottant. La reconfiguration des matrices cellulaires est contrôlée par un mot de *3-bit* appelé *Opcode.* Une entrée de *3-bit* appelée *MuxCtrl* est employée pour le contrôle d'un multiplexeur. Les sorties de *32-bit* du processeur sont respectivement appelées *Output0* et *Output1*. Le code Verilog-HDL suivant est utilisé pour déclarer cette fonction et ses ports d'entrées et sorties.

```
module UPM_ARRAYS ( FP_clock, Opcode, MuxCtrl, Samples, Samples1, Output0, Output1);
        //Déclaration des ports d'entrées et sorties
        input              FP_clock;          //Horloge des opérateurs points flottant
        input   [31:0]  Samples, Samples1;  //Entrée du processeur matriciel
        input   [2:0]    Opcode;              //Contrôle la reconfiguration
        input   [2:0]    MuxCtrl;          //Contrôle de multiplexeur
        output  [31:0]  Output0, Output1;     // Sorties du processeur matriciel
        reg     [31:0]  Output0, Output1;    //Sorties du processeur matriciel
```

Tout le programme de cette matrice 2x2 figure dans les ANNEXES 13 à 18, cependant des parties sont reprises dans ce qui suit afin d'expliquer, avec plus de détails, la conception des matrices. Ainsi les données en entrées sont reçues par l'intermédiaire des ports d'entrées *Samples* et *Samples1*, et dépendamment de la valeur du mot *Opcode*, une série de registres à décalage est utilisée pour acheminer ces données vers la matrice d'UPMs qui effectue les calculs et transmet les résultats aux sorties *Output0* et *Output1* en utilisant un multiplexeur pour les fonctions dont les résultats sont nombreux. La figure 3-5 et le tableau 5 montrent, pour chaque reconfiguration représentant une valeur différente d'*Opcode*, les sous-fonctions utilisées pour acheminer les

données des entrées périphériques vers les cellules UPMs et celles utilisées pour envoyer les résultats des cellules UPMs vers les ports de sorties du processeur. Ainsi à chaque changement de valeur du mot *Opcode* les ports d'entrées et sorties périphériques sont reconnectés aux connexions des cellules à l'aide de séries de registres et d'un multiplexeur dépendamment de la fonction à traiter. Deux séries de registres sont utilisées, l'une dont les registres sont connectés entre eux et l'autre dont les registres sont indépendants les uns des autres. Le programme suivant décrit la déclaration des registres et de leurs interconnexions :

```
///Interconnexions des registres
wire    [31:0]  data1, data2, data3, data4, data5, data6, data7, data8;
wire    [31:0]  data9, data10, data11, data12, data13, data14, data15;
wire    [31:0]  datax1, datax2, datax3, datax4, datax5, datax6, datax7, datax8;
wire    [31:0]          MuxOutput, data16, data17;

//Appels de fonctions registres de 32-bits indépendants entre eux
Registre32b     regA  (clk0, Samples, clk0, datax1);
Registre32b     regB  (clk, dataI2, clk, datax2);
Registre32b     regC  (clk, dataI3, clk, datax3);
Registre32b     regD  (clk, dataI4, clk, datax4);
Registre32b     regE  (clk, dataI5, clk, datax5);
Registre32b     regF  (clk, dataI6, clk, datax6);
Registre32b     regG  (clk, dataI7, clk, datax7);
Registre32b     regH  (clk, dataI8, clk, datax8);

//Appels de fonctions registres de 32-bits interconnectés entre eux
Registre32b     reg0  (clk0, Samples1, clk0, data1);
Registre32b     reg1  (clk0, data1, clk0, data2);
Registre32b     reg2  (clk0, data2, clk0, data3);
Registre32b     reg3  (clk0, data3, clk0, data4);
Registre32b     reg4  (clk0, data4, clk0, data5);
Registre32b     reg5  (clk0, data5, clk0, data6);
Registre32b     reg6  (clk0, data6, clk0, data7);
Registre32b     reg7  (clk0, data7, clk0, data8);
Registre32b     reg8  (clk0, data8, clk0, data9);
Registre32b     reg9  (clk0, data9, clk0, data10);
Registre32b     reg10 (clk0, data10, clk0, data11);
Registre32b     reg11 (clk0, data11, clk0, data12);
Registre32b     reg12 (clk0, data12, clk0, data13);
Registre32b     reg13 (clk0, data13, clk0, data14);
Registre32b     reg14 (clk0, data14, clk0, data15);
Registre32b     reg15 (clk0, data15, clk0, data16);
Registre32b     reg16 (clk0, data16, clk0, data17);
```

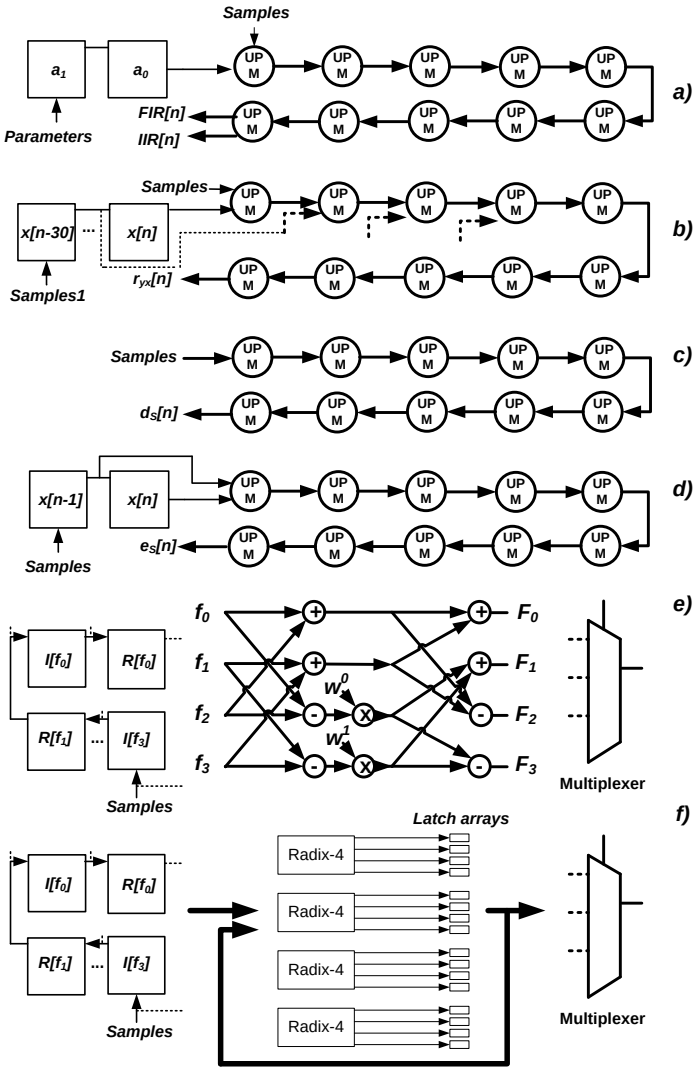


Figure 3-5 : Configuration des connexions pour différents algorithmes programmés.

Le programme suivant montre la déclaration des connexions internes de la fonction multiplexeur de *8*x*32*-bit et de son appel :

```
//Interconnexions des multiplexeurs
 reg    [31:0]  InMux0, InMux1, InMux2, InMux3;
reg    [31:0]  InMux4, InMux5, InMux6, InMux7;
wire    [31:0]   MuxOutput
//Appel d'un multiplexeur de 8x32b pour le contrôle des fonctions à plusieurs sorties
Multiplexeur8x32b    mx0 ( InMux0, InMux1, InMux2, InMux3, InMux4, InMux5, InMux6, InMux7, MuxCtrl, MuxOutput);
```

De même pour chaque mode de reconfiguration les connexions internes entre chaque cellule, constituant le processeur, sont reconfigurées. Les cellules sont reconnectées entre elles en changeant la valeur des bits de contrôle de la reconfiguration *Opcode*. Le code Verilog-HDL suivant présente la déclaration de toutes les connexions internes de chaque cellule du processeur.

```
//Internal registers
reg    [31:0]  InA2_c0, InA2_c1, InA2_c2, InA2_c3;
reg    [31:0]  InA1_c0, InA1_c1, InA1_c2, InA1_c3;
reg    [31:0]  InA0_c0, InA0_c1, InA0_c2, InA0_c3;
reg    [31:0]  InX5_c0, InX5_c1, InX5_c2, InX5_c3;
reg    [31:0]  InX2_c0, InX2_c1, InX2_c2, InX2_c3;
reg    [31:0]  InX1_c0, InX1_c1, InX1_c2, InX1_c3;
reg    [31:0]  InX0_c0, InX0_c1, InX0_c2, InX0_c3;
reg    [31:0]  InX4_c0, InX4_c1, InX4_c2, InX4_c3;
reg    [31:0]  InX3_c0, InX3_c1, InX3_c2, InX3_c3;
reg    [31:0]  InA4_c0, InA4_c1, InA4_c2, InA4_c3;
reg    [31:0]  InA3_c0, InA3_c1, InA3_c2, InA3_c3;
reg    [31:0]  InD2_c0, InD2_c1, InD2_c2, InD2_c3;
reg    [31:0]  InX6_c0, InX6_c1, InX6_c2, InX6_c3;
reg            bit3_c0, bit3_c1, bit3_c2, bit3_c3, bit1, ad0_c0, ad0_c1, ad0_c2, ad0_c3;
reg            ad1_c0, ad1_c1, ad1_c2, ad1_c3, ad2_c0, ad2_c1, ad2_c2, ad2_c3;
reg            ad3_c0, ad3_c1, ad3_c2, ad3_c3, ad4_c0, ad4_c1, ad4_c2, ad4_c3;
reg            ad5_c0, ad5_c1, ad5_c2, ad5_c3;
//Internal connexions
wire   [31:0]  Out0_c0, Out0_c1, Out0_c2, Out0_c3;
wire   [31:0]  Out1_c0, Out1_c1, Out1_c2, Out1_c3;
wire   [31:0]  Out2_c0, Out2_c1, Out2_c2, Out2_c3;
wire   [31:0]  Out3_c0, Out3_c1, Out3_c2, Out3_c3;
wire   [31:0]  Out4_c0, Out4_c1, Out4_c2, Out4_c3;
wire   [31:0]  Out5_c0, Out5_c1, Out5_c2, Out5_c3;
wire   [31:0]  Out6_c0, Out6_c1, Out6_c2, Out6_c3;
wire   [31:0]  result2_c0, result2_c1, result2_c2, result2_c3;
wire           S0, S1, S2, clk_Opcode0, bit2_Opcode0, clk0_Opcode0;
```

Comme illustrées sur la figure 3-6, les cellules UPMs sont respectivement appelées *c0, c1, c2 et c3*. Des extensions *_c0, _c1, _c2, ..., c_23* sont utilisées pour différencier les noms des connexions internes de chaque UPM des autres. Pour chaque appel de la fonction UPM les noms des connexions internes de chaque cellule correspondent à ceux illustrés sur la figure 3-6 comme montré dans le code suivant.

```
//UPM function calls
UPM  c0 (InA2_c0, InA1_c0,InA0_c0, InX5_c0, InX2_c0, InX1_c0, InX0_c0, InX4_c0, InX3_c0, InX6_c0, InA4_c0,  InA3_c0, InD2_c0, FP_clock, bit1, bit2, bit3_c0, ad0_c0, ad1_c0, ad2_c0, ad3_c0, ad4_c0,  ad5_c0, Out0_c0, Out1_c0, Out4_c0, Out3_c0, Out2_c0, Out5_c0, Out6_c0, result2_c0);
/
UPM  c1 (InA2_c1, InA1_c1,InA0_c1, InX5_c1, InX2_c1, InX1_c1, InX0_c1, InX4_c1, InX3_c1, InX6_c1, InA4_c1,  InA3_c1, InD2_c1, FP_clock, bit1, bit2, bit3_c1, ad0_c1, ad1_c1, ad2_c1, ad3_c1, ad4_c1,  ad5_c1, Out0_c1, Out1_c1, Out4_c1, Out3_c1, Out2_c1, Out5_c1, Out6_c1, result2_c1);

UPM  c2 (InA2_c2, InA1_c2,InA0_c2, InX5_c2, InX2_c2, InX1_c2, InX0_c2, InX4_c2, InX3_c2, InX6_c2, InA4_c2,  InA3_c2, InD2_c2, FP_clock, bit1, bit2, bit3_c2, ad0_c2, ad1_c2, ad2_c2, ad3_c2, ad4_c2,  ad5_c2, Out0_c2, Out1_c2, Out4_c2, Out3_c2, Out2_c2, Out5_c2, Out6_c2, result2_c2);

UPM  c3 (InA2_c3, InA1_c3,InA0_c3, InX5_c3, InX2_c3, InX1_c3, InX0_c3, InX4_c3, InX3_c3, InX6_c3, InA4_c3,  InA3_c3, InD2_c3, FP_clock, bit1, bit2, bit3_c3, ad0_c3, ad1_c3, ad2_c3, ad3_c3, ad4_c3,  ad5_c3, Out0_c3, Out1_c3, Out4_c3, Out3_c3, Out2_c3, Out5_c3, Out6_c3, result2_c3);
```

Des valeurs constantes sont utilisées comme paramètres préprogrammés et leurs déclarations sont montrés dans le code suivant.

```
//Initialisation des paramètres
parameter un      =  32'b00111111100000000000000000000000;  //=1
parameter moinsun =  32'b10111111100000000000000000000000;  //=-1
parameter zero    =  32'b00000000000000000000000000000000;  //=0
parameter two     =  32'b01000000000000000000000000000000;  //=2 ...
```

Tableau 5 : Mode de transfert des données pour chaque mode de reconfiguration.

Fonction	**Opcode**	**Mode de transfert des données en entrée**	**Mode de transfert des données en sortie**
Filtrage non-Adaptatif	000	-L'entrée *Samples* est connectée directement aux cellules. -L'entrée *Samples1* n'est pas utilisée car les paramètres du filtrage sont préprogrammés.	-La sortie FIR est connectée à *Output0.* -La sortie IIR est connectée à *Output1*.
Filtrage Adaptatif	001	-L'entrée *Samples* est connectée directement aux cellules. -L'entrée *Samples1* est connectée à une série de registre à décalage interconnectés.	La sortie FIR est connectée à *Output0.* -La sortie IIR est connectée à *Output1*.
All-Pole	010	-L'entrée *Samples* est connectée directement à l'entrée $\tilde{d}_S[n]$ cellules.	-Les sorties $d_{S-1}[n]$ et $\tilde{d}_{S-1}[n]$ sont connectées respectivement à *Output0* et *Output1*.
All-zero	011	-L'entrée *Samples* est connectée directement à l'entrée $x[n]$ des cellules. -L'entrée *Samples* est retardée de un *tap* par l'intermédiaire d'un registre puis le résultat $x[n-1]$est connecté directement aux cellules.	-Les sorties $e_S[n]$ et $\tilde{e}_S[n]$ sont connectées respectivement à *Output0* et *Output1*.
Operateur butterfly radical-4	100	-L'entrée *Samples* est connectée à une série de registre à décalage.	-À la fin du traitement les sorties FFT sont connectées à des registres indépendants qui enregistrent les données par la suite envoyées à un multiplexeur dont la sortie est connectée à *Output0* et contrôlée par *MuxCtrl.*
Division selon Newton-Raphson	101	-Les ports d'entrées sont connectés directement aux cellules	-Les sorties des cellules sont connectées directement aux ports de sorties
Division selon Newton-Raphson	110	-Les ports d'entrées sont connectés directement aux cellules	-Les sorties des cellules sont connectées directement aux ports de sorties

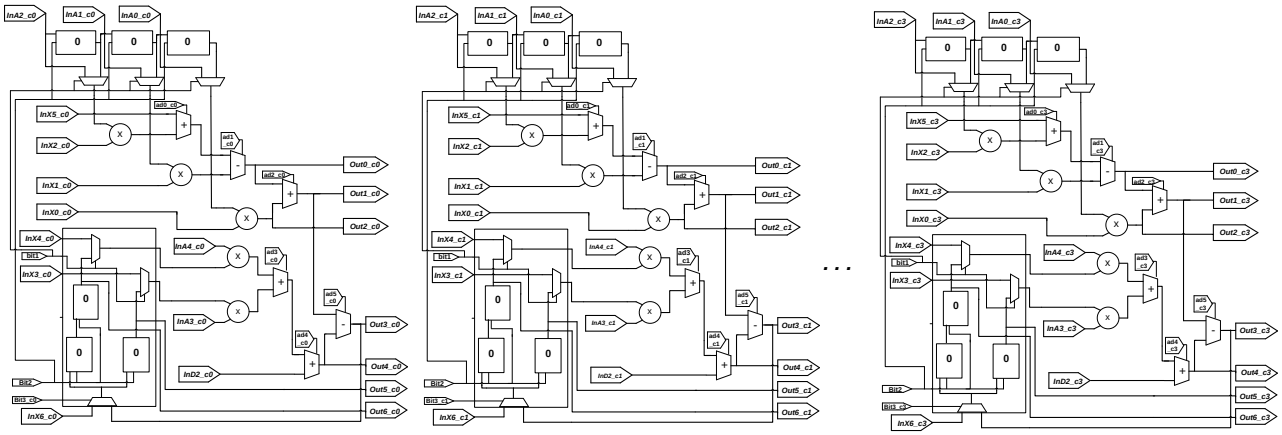


Figure 3-6 : Noms des ports d'entrées et sorties et des connexions internes des cellules.

3.2.2 Contrôle temporel du processeur matriciel

Dans le but de permettre un grand nombre d'opérations de traitement de signaux les connexions internes entre les UPMs sont reconfigurées. Pour chaque reconfiguration, un des algorithmes de configuration des connexions, illustrés dans la figure 3-5, est utilisé. Ces algorithmes utilisent quatre horloges de contrôle synchronisées, respectivement, *FP_clock* représentant l'horloge d'entrée dédiée aux opérateurs point flottant, *clk* l'horloge principale qui contrôle l'algorithme temporel et qui est générée à partir de *FP_clock* afin de contrôler le transfert de données entre chaque cellule du processeur matriciel. L'horloge *clk_bit2* appelée, *bit2* dans le programme, est générée et en phase avec *clk* afin de contrôler le calcul de toute la matrice de cellule, son utilisation est aussi employée pour contrôler le décalage des données à filtrer qui sont transférées entre les registres. L'horloge *clk0* est utilisée pour contrôler l'acquisition des paramètres du filtre par une série de registres. Cependant ce processeur matriciel ne prend qu'une seule horloge comme port d'entrée qui est *FP_clock*. Plusieurs algorithmes sont construits afin de générer, à partir de l'horloge *FP_clock*, les horloges *clk*, *clk_bit2* et *clk0*. Ainsi un programme appelé *clock_divider.v* est écrit pour diviser l'horloge *FP_clock* et ce code est programmé dans le paragraphe suivant. Ce programme permet de créer une nouvelle horloge synchronisée avec l'horloge d'entrée *FP_clock* en divisant celle-ci par un nombre égal à la valeur du mot *div0* avec un rapport cyclique égal à *div*/*div0*.

```
//Diviseur de l'horloge FP_clk
module clock_divider ( FP_clk, div, div0, clk);
input        FP_clk;
```

```
input  [4:0] div, div0;
output       clk;
wire         clk;
reg                  enable;
reg          [4:0] count;

assign clk = enable;
always @ ( posedge FP_clk)
begin
   count=count+1'b1;
                  if(count<div)
                        begin
                        enable=1'b1;
                        end
                  if(count>=div)
                        begin
                        enable=1'b0;
                        end
                  if(count==div0)
                        begin
                        count =5'b00000;
                        end
end

endmodule
```

Ce programme *clock_divider* est utilisé pour générer deux horloges *clk* et *clk_bit2* et ceci pour chaque reconfiguration avec des fréquences différentes lorsque cela est nécessaire. Un algorithme est aussi utilisé pour retarder les horloges avec un délai de plusieurs fois l'horloge d'entrée *clk*, ce délai a une valeur égale à la valeur du mot *del* comme le montre le code suivant :

```
//Programme pour retarder une horloge
module delay (clk, del, reset);
input        clk;
input  [6:0] del;
output       reset;
wire         reset;
reg                  enable;
reg          [6:0] i;

assign reset = clk & enable;

always @ ( posedge clk)
begin
i=i+1;
        if (i<=del)
        begin
        enable=1'b0;
        end
        else if (i>del)
        begin
        enable=1'b1;
        i=i-1;
```

```
        end
end
endmodule
```

Un autre algorithme est utilisé pour générer l'horloge *clk0* qui est utilisée afin de contrôler une série de registres pour l'acquisition de *32* paramètres de filtrage avant le calcul:

```
//Programme de division de l'horloge FP_clk
module clock_divider0 (clk, reset);
input       clk;
output      reset;
wire        reset;
reg                 enable;
reg         [5:0] i;

assign reset = clk & enable;

always @ ( posedge clk)
begin
i=i+1;
        if (i<=32)
        begin
        enable=1'b1;
        end
        else if (i>32)
        begin
        enable=1'b0;
        i=6'b100001;
        end
end
endmodule
```

Ce programme peut être préprogrammé en changeant la valeur du nombre des données à acquérir comme illustré dans le code suivant pour deux données :

```
//Programme de division de l'horloge FP_clk
module clock_divider1 (clk, reset);
input       clk;
output      reset;
wire        reset;
reg                 enable;
reg         [5:0] i;

assign reset = clk & enable;


always @ ( posedge clk)
begin
i=i+1;
        if (i<=2)
        begin
        enable=1'b1;
        end
```

```
        else if (i>2)
        begin
        enable=1'b0;
        i=6'b000100;
        end
end
endmodule
```

Ces fonctions de génération d'horloge sont appelées tel que ce qui suit :

```
//Construction des horloges de contrôle à partir de l'entrée FP_clock
delay          dl0   (FP_clock, 7'b1000001, S0);
clock_divider  cl0   (S0, 5'b01110, 5'b11010, clk_Opcode0);
delay          dl1   (clk_Opcode0, 7'b0000010, S1);
clock_divider  cl1   (S1, 5'b00101, 5'b01000, bit2_Opcode0);
clock_divider0    cl2   (FP_clock,   clk0_Opcode0);
clock_divider1    cl3   (FP_clock,   clk0_Opcode1);
```

Pour chaque reconfiguration et valeur du mot *Opcode* des horloges *clk*, *clk0* et *clk_bit2* sont générées et envoyées dans trois multiplexeurs contrôlés par le mot *Opcode*. Ces multiplexeurs de *1*x*8*-bits sont appelés de la manière suivante et ont respectivement pour sortie *clk*, *bit2* et *clk0* :

```
Multiplexeur6x1b     mx1   ( clk_Opcode0, clk_Opcode0, clk_Opcode0, clk_Opcode0, clk_Opcode0, clk_Opcode0,
                             Opcode,    clk);
Multiplexeur6x1b     mx2   ( bit2_Opcode0, bit2_Opcode0, bit2_Opcode0, bit2_Opcode0, bit2_Opcode0,
                             bit2_Opcode0, Opcode,    bit2);
Multiplexeur6x1b     mx3   ( clk0_Opcode0, clk0_Opcode0, clk0_Opcode0, clk0_Opcode0, clk0_Opcode0,
                             clk0_Opcode0, Opcode,   clk0);
```

3.2.3 La reconfiguration des connexions internes entre les cellules

Dans le but de permettre un grand nombre d'opérations de traitement de signaux les connexions internes entre les UPMs sont reconfigurées. Un algorithme de connexions préprogrammées est utilisé pour chaque configuration comme illustré dans la figure 3-5. Ainsi à chaque montée de l'horloge temporelle *clk* le choix de la reconfiguration est contrôlé par le mot *Opcode*. Si *Opcode* est égal à la valeur de *0000b* en binaire, un vecteur de dix UPMs opèrent en corrélateur et filtrage non adaptatif FIR/IIR d'ordre *n*=9. Si *Opcode* est égal à la valeur de *0001b*, le même vecteur est configuré en mode filtrage adaptatif FIR/IIR d'ordre *n*=9. Si *Opcode* est égal à *0010b* les UPMs sont reconfigurés en filtre en treillis All-Pole d'ordre *s*=4. Si *Opcode* est égal à *0011b*, le processeur vectoriel est reconfiguré en filtre en treillis All-Zero d'ordre *s*=4. Si *Opcode* est égal à *0100b*, cette matrice cellulaire est reconfigurée pour opérer en FFT radical-4. Tous ces algorithmes sont contenus dans une boucle de programmation contrôlée par *clk* sont programmés

dans les instructions Verilog-HDL suivantes présentées comme une boucle à l'intérieur de laquelle sont reconfigurées des connexions internes pour chaque valeur du mot *Opcode* :

```
always @ (posedge clk)
begin

////

end
```

À l'intérieur de cette boucle sont écrites les reconfigurations des connexions pour chaque mode de fonctionnement qui sont présentés dans les points suivants :

- Filtrage Non Adaptatif

La configuration filtrage non adaptatif est illustrée sur les figures 2-10 à 2-14 et les mêmes connexions entre les cellules qui y figurent sont programmées dans le code Verilog-HDL du paragraphe suivant :

```
//////////////////////////////
/////NON-ADAPTIVE
FIR-IIR MODE N=9//
//////////////////////////////
 if (Opcode==3'b000)
begin
//NON-ADAPTIVE
//FIR-IIR
//processor input
InA2_c0 <= Samples;
//Control bits
bit1 <= 1'b0;
bit3_c0 <= 1'b1;
bit3_c1 <= 1'b1;
bit3_c2 <= 1'b1;
bit3_c3 <= 1'b1;
ad0_c0 <= 1'b1;
ad1_c0 <= 1'b1;
ad2_c0 <= 1'b1;
ad3_c0 <= 1'b1;
ad4_c0 <= 1'b1;
ad5_c0 <= 1'b0;
ad0_c1 <= 1'b1;
ad1_c1 <= 1'b1;
ad2_c1 <= 1'b1;
ad3_c1 <= 1'b1;
ad4_c1 <= 1'b1;
ad5_c1 <= 1'b0;
ad0_c2 <= 1'b1;
ad1_c2 <= 1'b1;
ad2_c2 <= 1'b1;
ad3_c2 <= 1'b1;
ad4_c2 <= 1'b1;
ad5_c2 <= 1'b0;
ad0_c3 <= 1'b1;
ad1_c3 <= 1'b1;
ad2_c3 <= 1'b1;
ad3_c3 <= 1'b1;
ad4_c3 <= 1'b1;
ad5_c3 <= 1'b0;
 //Initialization of
//UPM c0 parameters
InA1_c0 <= zero;
InA0_c0 <= zero;
InX5_c0 <= zero;
InX2_c0 <= un;
InX1_c0 <= un;
InX0_c0 <= un;
InX4_c0 <= zero;
InX3_c0 <= zero;
InA4_c0 <= un;
InA3_c0 <= un;
InD2_c0 <= zero;
 //Interconnexion //with
neighborhood //UPM
InX6_c0 <= Out3_c3;
InX5_c1 <= Out1_c0;
InD2_c1 <= Out4_c0;
InX6_c1 <= Out6_c0;
//Initialization of //UPM
c1 parameters
InA2_c1<= result2_c0;
InA1_c1<= zero;
InA0_c1 <= zero;
InX2_c1 <= un;
InX1_c1 <= un;
InX0_c1 <= un;
InX4_c1 <= zero;
InX3_c1 <= zero;
InA4_c1 <= un;
InA3_c1 <= un;
 //Interconnexion //with
neighborhood UPM
InX5_c2 <= Out1_c1;
InD2_c2 <= Out4_c1;
InX6_c2 <= Out6_c1;
 //Initialization of //UPM
c2 parameters
InA2_c2 <= zero;
InA1_c2 <= zero;
InA0_c2 <= zero;
InX2_c2 <= zero;
InX1_c2 <= zero;
InX0_c2 <= zero;
InX4_c2 <= zero;
InX3_c2 <= zero;
InA4_c2 <= un;
InA3_c2 <= un;
 //Interconnexion //with
neighborhood UPM
InX5_c3 <= Out1_c2;
InD2_c3 <= Out4_c2;
InX6_c3 <= Out6_c2;
 //Initialization of
//UPM c3 parameters
InA2_c3<= result2_c1;
InA1_c3 <= zero;
InA0_c3 <= zero;
InX2_c3 <= un;
InX1_c3 <= un;
InX0_c3 <= un;
InX4_c3 <= zero;
InX3_c3 <= zero;
InA4_c3 <= un;
InA3_c3 <= un;
 //FIR IIR Array
//processor outputs
Output0 <= Out1_c3;
 //IIR output delayed
//by one
 Output1 <= Out3_c3;
end
```

Cette configuration est employée lorsque le mot *Opcode* est égal à la valeur de *0000b* et pour les fonctions non adaptives, c'est-à-dire que les paramètres du filtre sont préprogrammés dans le programme. La figure 3-6 illustre les noms des connexions internes utilisées dans ce code pour

connecter les cellules entre elles. L'entrée *Samples* du processeur vectoriel est connectée à *InA2_c0* qui représente une entrée de la cellule *c0.* Les paramètres des filtres représentés par les connexions *InX2, InX1, InX0, InA3* et *InA4* sont fixés à une valeur égale à *1* pour des questions de simplicité. Les cellules sont connectées entre elles par l'intermédiaire des ports *InX5, InD2* et *InX6* et par ceci à chaque montée d'horloge de *clk,* un transfert de données est effectué entre chaque cellule. C'est-à-dire que le résultat du calcul qui est effectué à l'intérieur de chaque cellule est transmis à la cellule suivante à chaque coup d'horloge de *clk*. Le résultat de la cellule *c0* va ainsi être transféré à la cellule *c1* au premier coup d'horloge de *clk,* de la cellule *c1* à la cellule *c2* au second coup d'horloge, de la cellule *c2* à *c3* au dernier coup d'horloge par l'intermédiaire des connexions internes *Out1* et *Out4*. Les connexions *Out1* et *Out4* représentent respectivement la somme cumulative des résultats FIR et IIR. Il est important de noter que chaque résultat IIR correspondant à la sortie *Out3_c3* issue d'une seule soustraction, des valeurs de *Out1_c3* et de *Out4_c3,* qui est effectuée par la dernière cellule. Les résultats FIR et IIR sortent des connexions *Out1_c3* et *Out3_c3* qui sont respectivement connectés à *Output0* et *Output1*, les sorties du processeur vectoriel.

La figure 3-3 illustre le nombre de multiplieurs et additionneurs point flottant utilisés pour une seule cellule. L'additionneur point flottant a été généré par l'environnement de programmation et d'après les spécifications techniques [51], un délai de 7 coups d'horloges *FP_clock* au minimum est requis pour terminer le calcul d'une addition de même que le multiplieur point flottant nécessite un délai de 5 coups d'horloges pour effectuer le calcul. On en déduit qu'il faut une profondeur maximale de 26 coups d'horloges pour qu'une seule cellule effectue un calcul. Ainsi l'horloge *clk* est fixée à 26 coups d'horloge de *FP_ clock* afin de contrôler le transfert des données entre chaque cellule du processeur matriciel. La figure 3-7 montre la configuration des horloges *FP_clock, clk, clk_bit2* et *clk0* utilisées pour ce mode. L'horloge *bit2* est générée et en phase avec *clk* afin de contrôler le calcul de toute la matrice de cellule. Un vecteur de 4 cellules configuré en FIR/IIR va donc prendre au minimum 4x26 coups d'horloges de *clk* tel qu'illustré dans la figure 3-8. Cependant *clk_bit2* a été fixée à 8 coups d'horloges pour laisser le temps à la simulation d'afficher clairement le résultat. L'horloge *clk_bit2* contrôle le calcul de 4 cellules par variations de données communément appelées "*data control*".

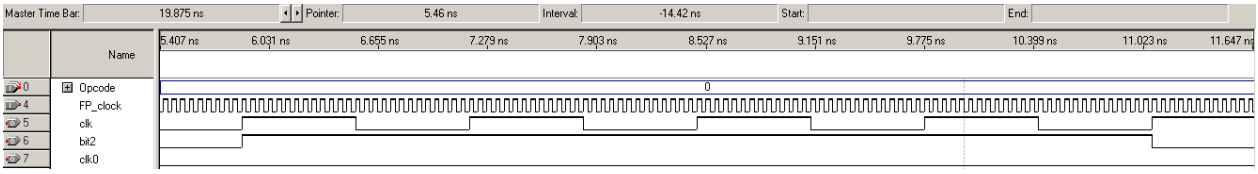


Figure 3-7 : Diagramme temporel des horloges de contrôle du mode FIR/IIR non adaptatif.

La figure 3-8 exprime pour la configuration FIR/IIR la manière dont les résultats des calculs de chaque cellule sont transférés de cellule en cellule à chaque coup d'horloge de *clk* par l'intermédiaire de leurs sorties *Out1* et *Out4*. L'horloge *clk_bit2* monte à la fin du calcul de toutes les cellules, soit au minimum de 4x26 coups d'horloges. Cette même horloge *clk_bit2* est utilisée pour contrôler les *taps*, ou transfert de données entre les registres internes, des UPMs car il faut attendre la fin des calculs de tout le vecteur de cellule avant de décaler les données dans le cas du filtrage IIR.

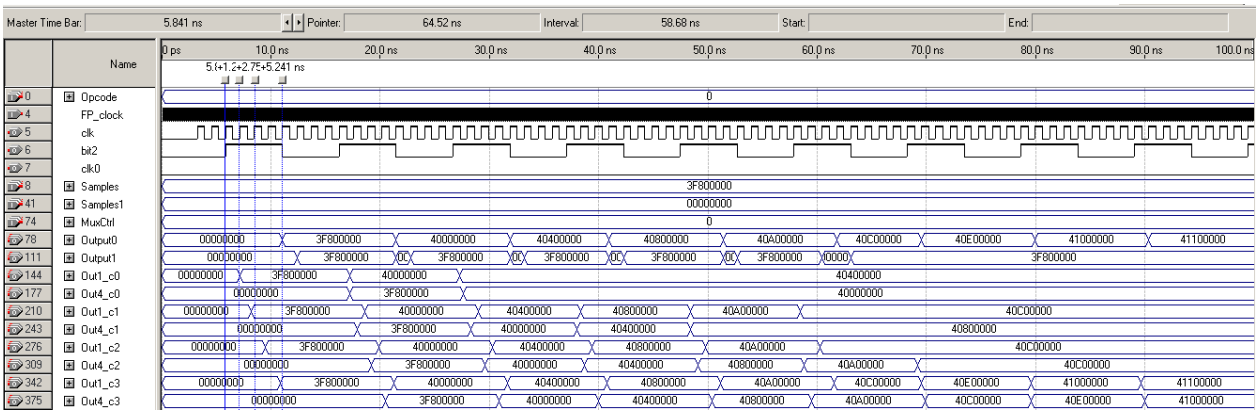


Figure 3-8 : Contrôle temporel du mode FIR/IIR non adaptatif.

Le mode IIR nécessite une récursivité entre les cellules c'est-à-dire que les cellules fonctionnent en boucle fermée et ceci explique qu'il faut que *clk_bit2* soit égale à une fréquence *clk* divisée par une valeur égale au nombre de cellules que compte le vecteur :

$$f_{bit2} = f_{clk} / (nb\ de\ cellule) \qquad (3.1)$$

- Filtrage Adaptatif

Cette reconfiguration est appliquée si le mot *Opcode* est égal à *001b*. La configuration filtrage adaptatif peut être utilisée pour le calcul de la fonction FIR/IIR qui est illustrée sur les figures 2-10 à 2-14, pour des fonctions de Corrélations montrées dans la figure 2-21, pour des calculs de la transformée de Hilbert et de Cosinus discrète représentées par la figure 2-22, pour la transformée de Hartley discrète et la transformée de Hankel qui sont schématisées dans la figure 2-23, pour la génération de fonctions par les polynômes de Chebyshev représentée par l'équation (2.16) et aussi pour le calcul des polynômes de Chebyshev. Les mêmes connexions entre les cellules sont programmées dans le code Verilog-HDL du paragraphe ci-dessous :

```
////////////////////////////////
ADAPTIVE FIR-IIR &
CORRELATOR MODE
N=9////////////////////////////
  if (Opcode==3'b001)
begin
//ADAPTIVE FIR-IIR
//CORRELATOR N=9
// processor input
 InA2_c0 <= Samples;
//Control bits
bit1        <= 1'b0;
bit3_c0 <= 1'b1;
bit3_c1 <= 1'b1;
bit3_c2 <= 1'b1;
bit3_c3 <= 1'b1;
ad0_c0  <= 1'b1;
ad1_c0  <= 1'b1;
ad2_c0  <= 1'b1;
ad3_c0  <= 1'b1;
ad4_c0  <= 1'b1;
ad5_c0  <= 1'b0;
ad0_c1  <= 1'b1;
ad1_c1  <= 1'b1;
ad2_c1  <= 1'b1;
ad3_c1  <= 1'b1;
ad4_c1  <= 1'b1;
ad5_c1  <= 1'b0;
ad0_c2  <= 1'b1;
ad1_c2  <= 1'b1;
ad2_c2  <= 1'b1;
ad3_c2  <= 1'b1;
ad4_c2  <= 1'b1;
ad5_c2  <= 1'b0;
ad0_c3  <= 1'b1;
ad1_c3  <= 1'b1;
ad2_c3  <= 1'b1;
ad3_c3  <= 1'b1;
ad4_c3  <= 1'b1;
ad5_c3  <= 1'b0;
//Initialization of
//UPM c0 parameters
InA1_c0  <=  zero;
InA0_c0 <= zero;
InX5_c0 <= zero;
InX2_c0 <= data1;
InX1_c0 <= data2;
InX0_c0 <= data3;
InX4_c0 <= zero;
InX3_c0 <= zero;
InA4_c0 <= data4;
InA3_c0 <= data5;
InD2_c0 <= zero;
 //Interconnexion
//with        neighborhood
//UPM
/InX6_c0 <= Out3_c3;
/InX5_c1 <= Out1_c0;
/InD2_c1 <= Out4_c0;
/InX6_c1 <= Out6_c0;
/ //Initialization
 //of UPM c1 parameters
InA2_c1 <= result2_c0;
InA1_c1  <= zero;
InA0_c1 <= zero;
InX2_c1 <= data6;
InX1_c1 <= data7;
InX0_c1 <= data8;
InX4_c1 <= zero;
InX3_c1 <= zero;
InA4_c1 <= data9;
InA3_c1 <= data10;
//Interconnexion
//with neighborhood
UPM
InX5_c2 <= Out1_c1;
InD2_c2 <= Out4_c1;
 InX6_c2 <= Out6_c1;
  //Initialization
//of UPM c2 parameters
InA2_c2 <= zero;
InA1_c2  <= zero;
InA0_c2 <= zero;
InX2_c2 <= zero;
InX1_c2 <= zero;
InX0_c2 <= zero;
InX4_c2 <= zero;
InX3_c2 <= zero;
InA4_c2 <= data11;
InA3_c2 <= data12;
  //Interconnexion
//with neighborhood
//UPM
InX5_c3 <= Out1_c2;
 InD2_c3 <= Out4_c2;
InX6_c3 <= Out6_c2;
 //Initialization of
  //UPM c3 parameters
InA2_c3 <= result2_c1;
InA1_c3  <= zero;
InA0_c3 <= zero;
InX2_c3 <= data13;
InX1_c3 <= data14;
InX0_c3 <= data15;
InX4_c3 <= zero;
InX3_c3 <= zero;
InA4_c3 <= data16;
InA3_c3 <= data17;
    //FIR IIR Array
//processor outputs
   Output0 <= Out1_c3;
   //IIR output delayed
//by one
   Output1 <= Out3_c3;
end
```

Ces connexions sont similaires à celles décrites pour le filtrage non adaptatif, dans le point précédent, car les données à filtrer sont envoyées vers les cellules par l'intermédiaire de l'entrée *Samples* connectée à *InA2_c0*. Cependant, à la différence du filtrage non adaptatif, des registres à décalages sont utilisés pour acheminer les paramètres adaptatifs du filtre, à partir des ports d'entrées périphériques vers le vecteur de cellules. Une série de registres à décalage dont l'entrée

est *Samples1* est utilisée, les sorties de ces registres sont au nombre de dix-sept et respectivement appelées *data1*, *data2*, …, *data17* comme programmées dans ce qui suit :

```
Registre32b     reg0  (clk0, Samples1, clk0, data1);
...
Registre32b     reg16 (clk0, data16, clk0, data17);
```

Ainsi l'horloge *clk0*, contrôle le temps d'acquisition des paramètres des filtres qui entrent par l'intermédiaire de la deuxième entrée *Samples1* dont les valeurs sont stockées dans une série de registres. Ces valeurs sont transférées vers les cellules par l'intermédiaire des ports *InX0*, *InX1*, *InX2*, *InA3* et *InA4* pour chaque cellule. Ceci permet de modifier la valeur des paramètres du filtrage en temps réel. La figure 3-9 montre la configuration de l'horloge *clk0* qui permet d'acquérir 32 données *Sampes1* et logiquement cette acquisition se fait avant le début des calculs. Ceci explique pourquoi les horloges *clk* et *clk_bit2* ont été retardées, pour éviter que le calcul ne commence sans l'acquisition de tous les paramètres. Comme illustrée sur la figure 3-10 l'entrée *Samples1* est utilisée pour l'acquisition des paramètres et l'entrée *Samples* pour l'acquisition des données à traiter.

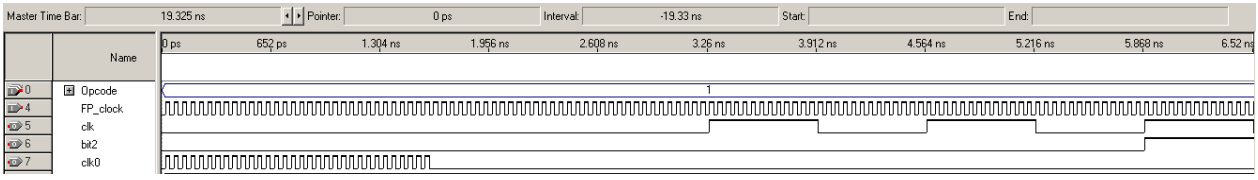


Figure 3-9 : Diagramme temporel des horloges de contrôle du mode FIR/IIR adaptatif.

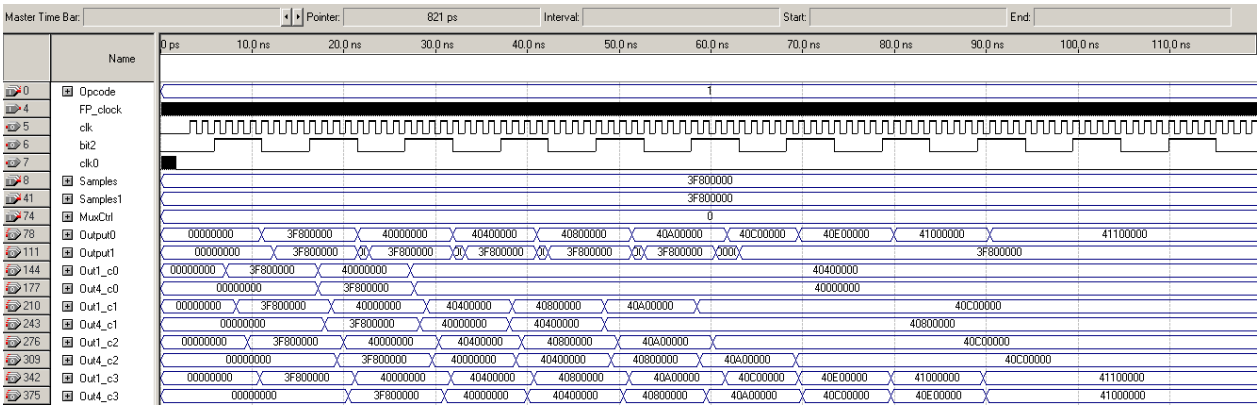


Figure 3-10 : Contrôle temporel du mode FIR/IIR adaptatif.

- Filtrage All-Pole

Cette configuration, appliquée lorsque *Opcode* est égal à *010b*, est illustrée à la figure 2-19 et les connexions configurées sont les mêmes que celles utilisés dans le code Vérilog-HDL du paragraphe suivant pour reconnecter les cellules entre elles :

```
//////////////////////////////
/////ALL-POLE
LATTICE MODE///
//////////////////////////////
else  if
(Opcode==3'b010)
begin
 //ALL-POLE LATTICE
 //Processor input
 InX2_c0 <= Samples;
//Control bits
bit1 <= 1'b1;
bit3_c0 <= 1'b1;
bit3_c1 <= 1'b1;
bit3_c2 <= 1'b1;
bit3_c3 <= 1'b1;
//Add/Sub bits
   ad0_c0  <= 1'b1;
   ad1_c0  <= 1'b0;
   ad2_c0  <= 1'b1;
   ad3_c0  <= 1'b1;
   ad4_c0  <= 1'b1;
   ad5_c0  <= 1'b0;
   ad0_c1  <= 1'b1;
   ad1_c1  <= 1'b0;
   ad2_c1  <= 1'b1;
   ad3_c1  <= 1'b1;
   ad4_c1  <= 1'b1;
   ad5_c1  <= 1'b0;
   ad0_c2  <= 1'b1;
   ad1_c2  <= 1'b0;
   ad2_c2  <= 1'b1;
   ad3_c2  <= 1'b1;
   ad4_c2  <= 1'b1;
   ad5_c2  <= 1'b0;
   ad0_c3  <= 1'b1;
   ad1_c3  <= 1'b0;
   ad2_c3  <= 1'b1;
   ad3_c3  <= 1'b1;
   ad4_c3  <= 1'b1;
   ad5_c3  <= 1'b0;
   //Initialization of
   //UPM c0 parameters
InA2_c0 <= un;
InA1_c0  <= un;
InA0_c0 <= zero;
InX5_c0 <= zero;
InX0_c0 <= zero;
   InA4_c0 <= un;
   InA3_c0 <= un;
   InD2_c0 <= zero;
   //Interconnexion
   //with neighborhood
   InX1_c0 <= Out5_c0;
   InX4_c0 <= Out5_c0;
   InX3_c0 <= Out0_c0;
   InX6_c0 <= Out4_c1;
   //Next UPM input
   InX2_c1 <= Out0_c0;
 //Initialization of
 //UPM c1 parameters
 InA2_c1 <= un;
 InA1_c1 <= un;
 InA0_c1 <= zero;
 InX5_c1 <= zero;
 InX0_c1 <= zero;
 InA4_c1 <= un;
 InA3_c1 <= un;
 InD2_c1 <= zero;
 //Interconnexion
 //with neighborhood
//inputs
   InX1_c1 <= Out5_c1;
   InX4_c1 <= Out5_c1;
   InX3_c1 <= Out0_c1;
   InX6_c1 <= Out4_c2;
   //Next UPM input
   InX2_c2 <= Out0_c1;
   //Initialization of
   //UPM c2 parameters
   InA2_c2 <= un;
 InA1_c2 <= un;
   InA0_c2 <= zero;
   InX5_c2 <= zero;
   InX0_c2 <= zero;
   InA4_c2 <= un;
   InA3_c2 <= un;
   InD2_c2 <= zero;
    //Interconnexion
    //with neighborhood
   InX1_c2 <= Out5_c2;
   InX4_c2 <= Out5_c2;
   InX3_c2 <= Out0_c2;
   InX6_c2 <= Out4_c3;
   //Next UPM input
   InX2_c3 <= Out0_c2;
   //Initialization of
   //UPM c2 parameters
InA2_c3 <= un;
InA1_c3  <= un;
   InA0_c3 <= zero;
   InX5_c3 <= zero;
   InX0_c3 <= zero;
   InA4_c3 <= un;
   InA3_c3 <= un;
   InD2_c3 <= zero;
//Interconnexion
//with neighborhood
//inputs
   InX1_c3 <= Out5_c3;
   InX4_c3 <= Out5_c3;
   InX3_c3 <= Out0_c3;
   InX6_c3 <= Out0_c3;
   //ALL-POLE
//LATTICE
   //processor output
   Output0 <= Out0_c3;
   Output1 <= Out4_c3;
end
```

Tel qu'illustré dans 2-19, l'entrée de données d_S[n] est représentée par la connexion *InX2_c0* qui est connectée au port d'entrée périphérique du processeur nommé *Samples*. La programmation des connexions entre les cellules suit le même patron de la figure 2-19. Ainsi le résultat d_{S-1}[n] de la cellule *c0* représenté par *Out0_c0* est connecté à l'entrée *InX2_c1* de la cellule *c1* et à l'entrée *InX3_c0* de la cellule *c0* de manière récursive; de même de que le résultat d_{S-2}[n] est connecté à l'entrée *InX2_c2* de la cellule *c2* et à *InX3_c1*; d_{S-3}[n] à *InX2_c3* de la cellule *c3* et à *InX3_c2* de manière récursive. En effet pour chaque cellule, les résultats respectifs $\tilde{d}_S$[n], $\tilde{d}_{S-1}$[n], … et d_{S-3}[n] sont retardés, par l'intermédiaire de l'entrée *InX6* du module de récursivité de la cellule précédente, donnant respectivement $\tilde{d}_S$[n-1], $\tilde{d}_{S-1}$[n-1], … et d_{S-3}[n-1] représentés par les sorties *Out5* de chaque cellule. Cette sortie *Out5* est connectée récursivement aux entrées *InX1* et *InX4*

pour chaque cellule comme illustré dans 2-19. La figure 3-11 illustre les horloges de contrôle de la configuration filtrage All-Pole et la figure 3-12 montre le diagramme temporel de cette configuration.

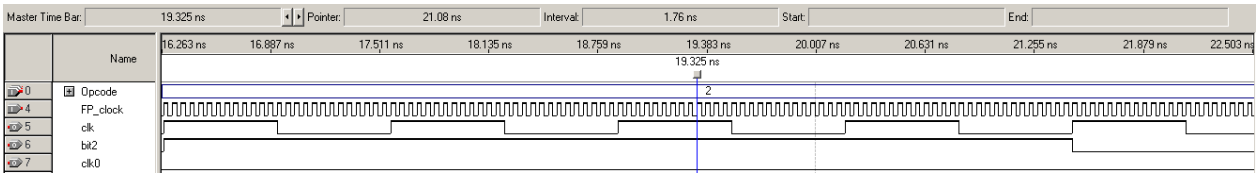


Figure 3-11 : Diagramme temporel des horloges de contrôle du filtre en treillis All-Pole.

La configuration des horloges est la même que celle utilisée pour le mode filtrage FIR/IIR car il y a une boucle fermée autour de la matrice. L'horloge *clk* qui contrôle le transfert des données entre chaque cellule est fixée à 26 coups de *FP_ clock* et *clk_bit2* qui est égal à 4*26 coups de *FP_clock* contrôle le calcul d'un vecteur de 4 cellules par variations de données communément appelées "*data control*". Des registres à décalage peuvent être utilisés pour rendre ce filtrage adaptatif en remplaçant dans le programme précédant les paramètres préprogrammés par des sorties de registres à décalage.

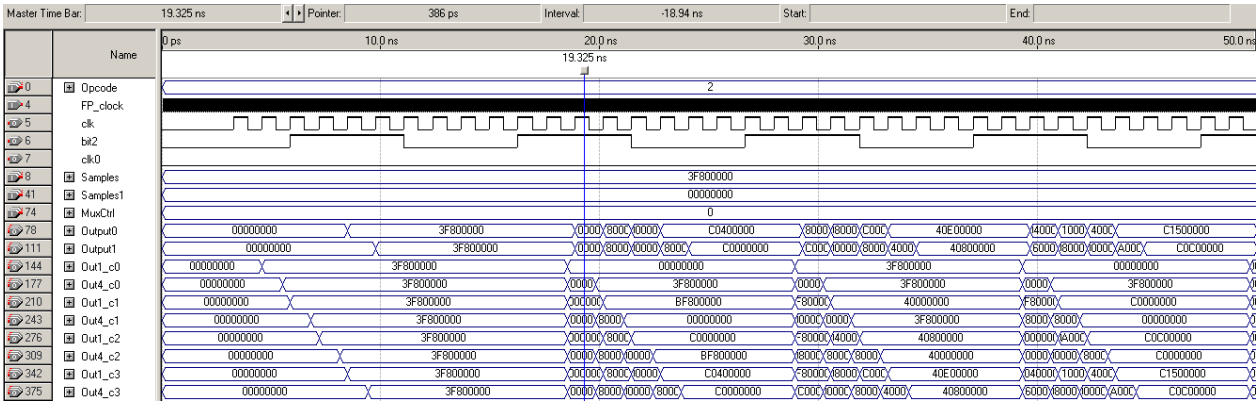


Figure 3-12 : Contrôle temporel du filtre en treillis All-Pole.

- Filtrage All-Zero

Lorsque le mot *Opcode* est égal à *011b* le processeur vectoriel est configuré en filtrage All-Zero qui est représenté par figures 2-16, 2-17 et 2-18. Comme programmée dans le code du paragraphe suivant, l'entrée périphérique du processeur vectoriel est *Samples* :

```
/////ALL-ZERO
LATTICE MODE///
//////////////////////////
 else  if
 (Opcode==3'b011)
begin
  //Processor inputs
    InX2_c0 <= Samples;
    InX1_c0 <= datax1;
    InX4_c0 <= datax1;
    InX3_c0 <= Samples;
//Control bits
    bit1 <= 1'b1;
    bit3_c0 <= 1'b1;
    bit3_c1 <= 1'b1;
    bit3_c2 <= 1'b1;
    bit3_c3 <= 1'b1;
//Add/Sub bits
    ad0_c0  <= 1'b1;
    ad1_c0  <= 1'b1;
    ad2_c0  <= 1'b1;
    ad3_c0  <= 1'b1;
    ad4_c0  <= 1'b1;
    ad5_c0  <= 1'b1;
    ad0_c1  <= 1'b1;
    ad1_c1  <= 1'b1;
    ad2_c1  <= 1'b1;
    ad3_c1  <= 1'b1;
    ad4_c1  <= 1'b1;
    ad5_c1  <= 1'b1;
    ad0_c2  <= 1'b1;
    ad1_c2  <= 1'b1;
    ad2_c2  <= 1'b1;
    ad3_c2  <= 1'b1;
    ad4_c2  <= 1'b1;
    ad5_c2  <= 1'b1;
    ad0_c3  <= 1'b1;
    ad1_c3  <= 1'b1;
    ad2_c3  <= 1'b1;
    ad3_c3  <= 1'b1;
    ad4_c3  <= 1'b1;
    ad5_c3  <= 1'b1;
    //Initialization of
    //UPM c0 parameters
  InA2_c0 <= un;
  InA1_c0 <= un;
    InA0_c0 <= zero;
    InX5_c0 <= zero;
    InX0_c0 <= zero;
    InA4_c0 <= un;
    InA3_c0 <= un;
    InD2_c0 <= zero;
    //Interconnexion
    //with neighborhood
    InX6_c0 <= Out4_c0;
    //Next UPM inputs
    InX2_c1 <= Out0_c0;
    InX3_c1 <= Out0_c0;
    InX1_c1 <= Out5_c0;
    InX4_c1 <= Out5_c0;
    //UPM c1 parameters
  InA2_c1 <= un;
  InA1_c1  <= un;
    InA0_c1 <= zero;
    InX5_c1 <= zero;
    InX0_c1 <= zero;
    InA4_c1 <= un;
    InA3_c1 <= un;
    InD2_c1 <= zero;
    //Interconnexion
    InX6_c1 <= Out4_c1;
    //Next UPM inputs
    InX2_c2 <= Out0_c1;
    InX3_c2 <= Out0_c1;
    InX1_c2 <= Out5_c1;
    InX4_c2 <= Out5_c1;
    //Initialization of
    //UPM c2 parameters
  InA2_c2 <= un;
  InA1_c2  <= un;
    InA0_c2 <= zero;
    InX5_c2 <= zero;
    InX0_c2 <= zero;
    InA4_c2 <= un;
    InA3_c2 <= un;
    InD2_c2 <= zero;
    //Interconnexion
    //with neighborhood
    InX6_c2 <= Out4_c2;
    //Next UPM inputs
    InX2_c3 <= Out0_c2;
    InX3_c3 <= Out0_c2;
    InX1_c3 <= Out5_c2;
    InX4_c3 <= Out5_c2;
    //UPM c3 parameters
  InA2_c3 <= un;
  InA1_c3  <= un;
    InA0_c3 <= zero;
    InX5_c3 <= zero;
    InX0_c3 <= zero;
    InA4_c3 <= un;
    InA3_c3 <= un;
    InD2_c3 <= zero;
    //Interconnexion
    //with neighborhood
    InX6_c3 <= Out4_c3;
    //ALL-ZERO
  LATTICE outputs
    Output0 <= Out0_c3;
    Output1 <= Out4_c3;
    end
```

Cette entrée qui représente *x*[n] est directement connectée aux entrées *InX2_c0* et *InX3_c0* de la cellule *c0*. En accord avec la figure 2-18 la même entrée *Samples* représentant *x*[n] est retardée en *x*[n-1] par l'intermédiaire d'un registre à décalage repris dans ce code :

```
 Registre32b     regA  (clk0, Samples, clk0, datax1);
```

La sortie de ce registre appelée *datax1* et représentant *x*[n-1] est connectée aux entrées *InX1* et *InX4* de la cellule *c0*. L'entrée *InX6* du module de récursivité est utilisée pour retarder le résultat $\tilde{e}_S[n]$, représenté par la sortie *Out4* de la même cellule. Le résultat de ce décalage appelé $\tilde{e}_S[n-1]$ et représenté par *Out5* est envoyé vers les entrées *InX1* et *InX4* de la cellule suivante. Les résultats du calcul du vecteur de cellules sont connectés à *Out0_c3* et *Out4_c3*.

Dans cette configuration il n'y a pas de boucle fermée entre le vecteur de cellules et seuls les registres internes des UPMs sont utilisés pour sauvegarder des résultats antérieurs. Le calcul sort à chaque coup d'horloge de *clk*. Ainsi l'horloge *clk_bit2* est mise égale à *clk*. Cette configuration

qui est égale à 26 coups d'horloge de *FP_clock* est montrée sur la figure 3-13 et prend le même temps de calcul que le mode FIR seul.

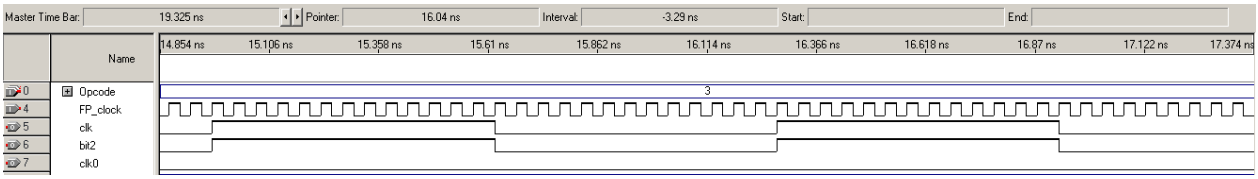


Figure 3-13 : Diagramme temporel des horloges de contrôle du filtre en treillis All-Zero.

L'horloge *clk0* peut être utilisée pour rendre le filtrage adaptatif comme le montre la figure 3-14 qui illustre le transfert de données entre les cellules.

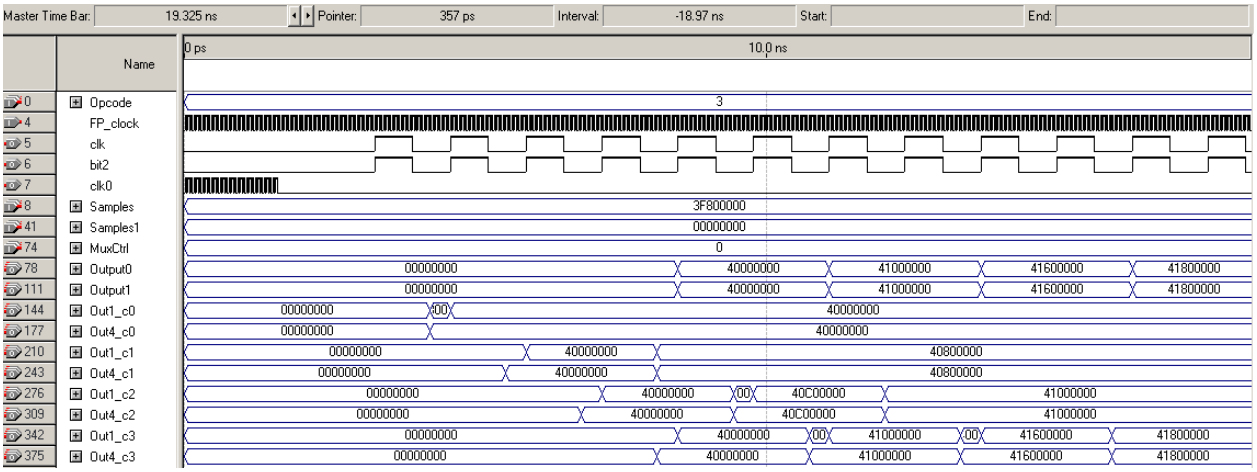


Figure 3-14 : Contrôle temporel du filtre en treillis All-Zero.

- Opérateur "*butterfly*" FFT radical-4

Les figures 2-26 et 2-27 a) schématisent respectivement la reconfiguration des mêmes cellules en fonction d'opérateur "*buttefly*" de transformée de Fourier radical-2 et radical-4 dont le programme est montré dans le paragraphe suivant :

////////////////////////// /////Radix-4 Butterfly/// ////////////////////////// else if

```
(Opcode==3'b100)
begin
//Butterfly r=4 mode
//Control bits//
bit1 <= 1'b1;
bit3_c0 <= 1'b0;
bit3_c1 <= 1'b0;
bit3_c2 <= 1'b0;
bit3_c3 <= 1'b0;
//C0 add_sub bits
ad0_c0  <= 1'b1;
   ad1_c0  <= 1'b1;
   ad2_c0  <= 1'b0;
   ad3_c0  <= 1'b1;
   ad4_c0  <= 1'b1;
   ad5_c0  <= 1'b1;
   ////C1 add_sub bit
   ad0_c1  <= 1'b1;
   ad1_c1  <= 1'b1;
   ad2_c1  <= 1'b0;
   ad3_c1  <= 1'b1;
   ad4_c1  <= 1'b1;
   ad5_c1  <= 1'b1;
   //C2 add_sub bit
   ad0_c2  <= 1'b0;
   ad1_c2  <= 1'b1;
   ad2_c2  <= 1'b0;
   ad3_c2  <= 1'b0;
   ad4_c2  <= 1'b1;
   ad5_c2  <= 1'b1;
   //C3 add_sub bit
   ad0_c3  <= 1'b0;
   ad1_c3  <= 1'b1;
   ad2_c3  <= 1'b0;
   ad3_c3  <= 1'b0;
   ad4_c3  <= 1'b1;
   ad5_c3  <= 1'b1;
//Configuration of
   // 2 UPMs c0 and c1
InA2_c0 <= un;
InA1_c0  <= un;
InA0_c0 <= two;
InX5_c0 <= data8;
 //R(f0)
InX2_c0 <= data4;
//R(f2)
InX1_c0 <= Out4_c0;
//R(s1)
InX0_c0 <= Out4_c0;
//R(s1)
InX4_c0 <= data6;
//R(f1)
InX3_c0 <= data2;
 //R(f3)
  InA4_c0 <= un;
  InA3_c0 <= un;
  InD2_c0 <= zero;
  InX6_c0 <= zero;
    //
InA2_c1 <= un;
InA1_c1  <= un;
InA0_c1 <= two;
InX5_c1 <= data7;
 //I(f0)
InX2_c1 <= data3;
  //I(f2)
InX1_c1   <=   Out4_c1;
//I(s1)
InX0_c1 <= Out4_c1;
//I(s1)
InX4_c1 <= data5;
 //I(f1)
 InX3_c1 <= data1;
 //I(f3)
InA4_c1 <= un;
InA3_c1 <= un;
InD2_c1 <= zero;
InX6_c1 <= zero;
//Configuration of
// 2 UPMs c2 and c3
InA2_c2 <= un;
InA1_c2  <= moinsun;
InA0_c2 <= moinstwo;
InX5_c2 <= data8;
 //R(f0)
InX2_c2 <= data4;
 //R(f2)
InX1_c2 <= Out4_c3;
//I(s3)
   InX0_c2 <= Out4_c3;
//I(s3)
   InX4_c2 <= data6;
//R(f1)
   InX3_c2 <= data2;
//R(f3)
   InA4_c2 <= un;
   InA3_c2 <= un;
   InD2_c2 <= zero;
   InX6_c2 <= zero;
   //
InA2_c3 <= un;
InA1_c3  <= un;
InA0_c3 <= two;
InX5_c3 <= data7;
//I(f0)
 InX2_c3 <= data3;
//I(f2)
InX1_c3 <= Out4_c2;
/R(s3)
InX0_c3 <= Out4_c2;
//R(s3)
   InX4_c3 <= data5;
//I(f1)
   InX3_c3 <= data1;
//I(f3)
   InA4_c3 <= un;
   InA3_c3 <= un;
   InD2_c3 <= zero;
   InX6_c3 <= zero;
   //r=4 Butterfly outputs
InMux0 <= Out0_c0;
//R(F0)
InMux1 <= Out0_c1;
//I(F0)
InMux2 <= Out1_c2;
//R(F1)
InMux3 <= Out1_c3;
//I(F1)
InMux4 <= Out1_c0;
//R(F2)
InMux5 <= Out1_c1;
//I(F2)
InMux6 <= Out0_c2;
//R(F3)
InMux7 <= Out0_c3;
//I(F3)
Output0 <= MuxOutput;
Output1 <= zero;
end
```

Dans cette configuration, la série de registres est utilisée pour acheminer les données qui entrent par l'intermédiaire du port d'entrée *Samples1*. Cette série de registres va fournir les données enregistrées par l'intermédiaire de ces connexions parallèles nommées *data1*, *data2*, …, *data8* représentants respectivement $R[f_0]$, $I[f_0]$, $R[f_1]$, $I[f_1]$, …, $I[f_3]$. Les connexions entre les cellules et les valeurs d'entrées sont programmées dans le code précédent en respectant précisément les connexions entre les cellules de la figure 2-27 a). Les résultats de cette configuration radical-4, qui sont appelés $R[F_0]$, $I[F_0]$, $R[F_1]$, $I[F_1]$, …, $I[F_3]$ sur la figure 2-27 a), sont acheminés vers le port de sortie *Output0* du processeur matriciel par l'intermédiaire d'un multiplexeur dont la sortie est nommée *MuxOutput* et ce multiplexeur est programmé dans les lignes suivantes :

```
Multiplexeur8x32b  mx0 ( InMux0, InMux1, InMux2, InMux3, InMux4, InMux5, InMux6, InMux7,
                         MuxCtrl, MuxOutput);
```

Le port d'entrée *Muxctrl* contrôle la sortie de ce multiplexeur comme illustré sur la figure 3-15. L'horloge *clk_bit2* n'est pas utilisée car la matrice n'est pas récursive et le calcul se fait de

manière parallèle. Le temps de calcul est de 3*26 coups de *FP_ clock*. L'horloge *clk0* contrôle l'acquisition des données par une série de registres avant le début des calculs.

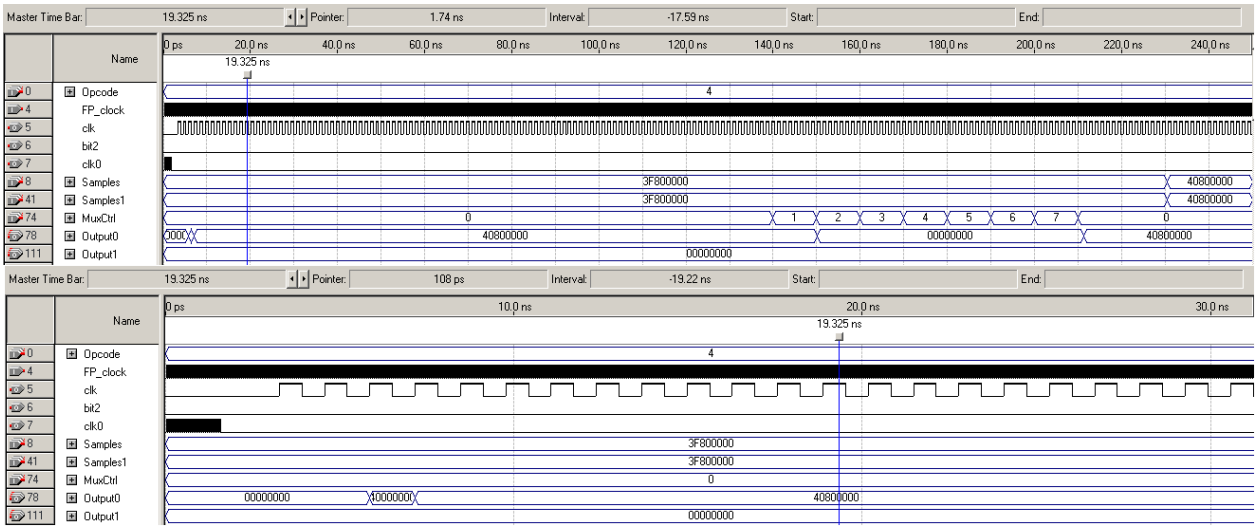


Figure 3-15 : Contrôle temporel de l'opérateur FFT radical 4.

- Algorithme récursif de division selon Newton-Raphson.

Cet algorithme est calculé lorsque la valeur de *Opcode* est égale à *101b* et son illustration est montrée dans la figure 2-33. Deux données respectivement nommées *B* et x_i sont utilisées et acheminées vers le processeur matriciel par l'intermédiaire des ports d'entrées *Samples* et *Samples1*. En effet *Samples1* est connectée à une série de registres à décalage dont *data2* est une des sorties qui sert à envoyer l'entrée x_i vers les connexions *InA2_ c0, InA0_ c0* et *InX0_ c0,* comme montré dans le programme ci-dessous:

```
//////////////////////////
/////DIVISION////////////
//////////////////////////
else                    if
(Opcode==3'b101)
begin
//Control bits
bit1 <= 1'b1;
bit3_c0 <= 1'b1;
bit3_c1 <= 1'b1;
bit3_c2 <= 1'b1;
bit3_c3 <= 1'b1;
//Add/Sub bits
   ad0_c0  <= 1'b1;
   ad1_c0  <= 1'b0;
   ad2_c0  <= 1'b1;
   ad3_c0  <= 1'b1;
   ad4_c0  <= 1'b1;
   ad5_c0  <= 1'b0;
   ad0_c1  <= 1'b1;
   ad1_c1  <= 1'b0;
   ad2_c1  <= 1'b1;
   ad3_c1  <= 1'b1;
   ad4_c1  <= 1'b1;
   ad5_c1  <= 1'b0;
   ad0_c2  <= 1'b1;
   ad1_c2  <= 1'b0;
   ad2_c2  <= 1'b1;
   ad3_c2  <= 1'b1;
   ad4_c2  <= 1'b1;
   ad5_c2  <= 1'b0;
   ad0_c3  <= 1'b1;
   ad1_c3  <= 1'b0;
   ad2_c3  <= 1'b1;
   ad3_c3  <= 1'b1;
   ad4_c3  <= 1'b1;
   ad5_c3  <= 1'b0;
//Initialization of
//UPM c0 parameters
InA2_c0 <= data2;
InA1_c0  <= datax1;
   InA0_c0 <= data2;
   InX5_c0 <= zero;
   InX2_c0 <= two;
   InX1_c0 <= Out2_c0;
   InX0_c0 <= data2;
   InA4_c0 <= zero;
   InA3_c0 <= zero;
   InD2_c0 <= zero;
   InX4_c0 <= zero;
   InX3_c0 <= zero;
   InX6_c0 <= zero;
//Initialization of
//UPM c1 parameters
InA2_c1 <= Out0_c0;
InA1_c1  <= datax1;
```

```
    InA0_c1 <= Out0_c0;
    InX5_c1 <= zero;
    InX2_c1 <= two;
    InX1_c1 <= Out2_c1;
    InX0_c1 <= Out0_c0;
    InA4_c1 <= zero;
    InA3_c1 <= zero;
    InD2_c1 <= zero;
    InX4_c1 <= zero;
    InX3_c1 <= zero;
    InX6_c1 <= zero;
//Initialization of
    //UPM c2 parameters
    InA2_c2 <= Out0_c1;
    InA1_c2  <= datax1;
       InA0_c2 <= Out0_c1;
       InX5_c2 <= zero;
       InX2_c2 <= two;
       InX1_c2 <= Out2_c2;
       InX0_c2 <= Out0_c1;
       InA4_c2 <= zero;
       InA3_c2 <= zero;
       InD2_c2 <= zero;
       InX4_c2 <= zero;
       InX3_c2 <= zero;
       InX6_c2 <= zero;
    //Initialization of
    //UPM c2 parameters
    InA2_c3 <= Out0_c2;
    InA1_c3  <= datax1;
       InA0_c3 <= Out0_c2;
       InX5_c3 <= zero;
       InX2_c3 <= two;
       InX1_c3 <= Out2_c3;
       InX0_c3 <= Out0_c2;
       InA4_c3 <= zero;
       InA3_c3 <= zero;
       InD2_c3 <= zero;
       InX4_c3 <= zero;
       InX3_c3 <= zero;
       InX6_c3 <= zero;
       //DIVISION
       //processor output
       Output0 <= Out0_c3;
       Output1 <= zero;
```

Le résultat *Out2* de chaque cellule qui représente la valeur de x_i^2 est connecté de manière récursive à l'entrée *InX1*. De même que le port d'entrée *Samples* est connecté à un registre indépendant dont la sortie est *datax1*. Cette sortie est utilisée pour acheminer la valeur de *B* vers l'entrée *InA1* de chaque cellule. Le résultat du calcul représenté par *Out0* qui est égal à x_{i+1} est reconnecté à la cellule suivante en accord avec la figure 2-33. Le résultat final *Out0_c3* est acheminé vers le port de sortie *Output0*. Le contrôle temporel de cette configuration est montré sur les figures 3-16 et 3-17. Chaque itération de calcul est représentée par un étage de cellule et ce vecteur de cellule n'est pas connecté en boucle fermée comme pour le filtrage FIR/IIR et ainsi l'horloge *clk_bit2* n'est pas utilisée et le temps de calcul est au maximum de 2*26 coups de *FP_clock*

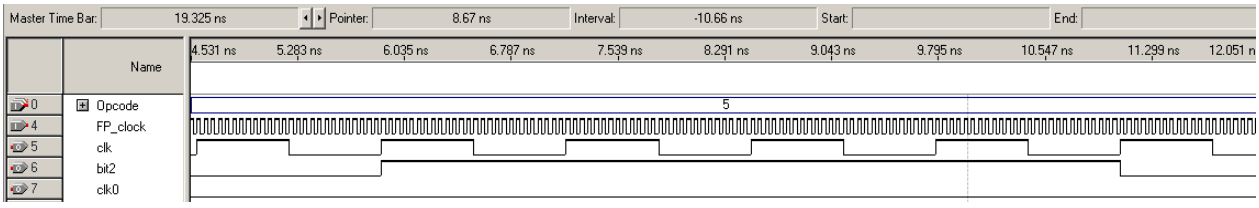


Figure 3-16 : Diagramme des horloges de l'algorithme de division selon Newton-Raphson.

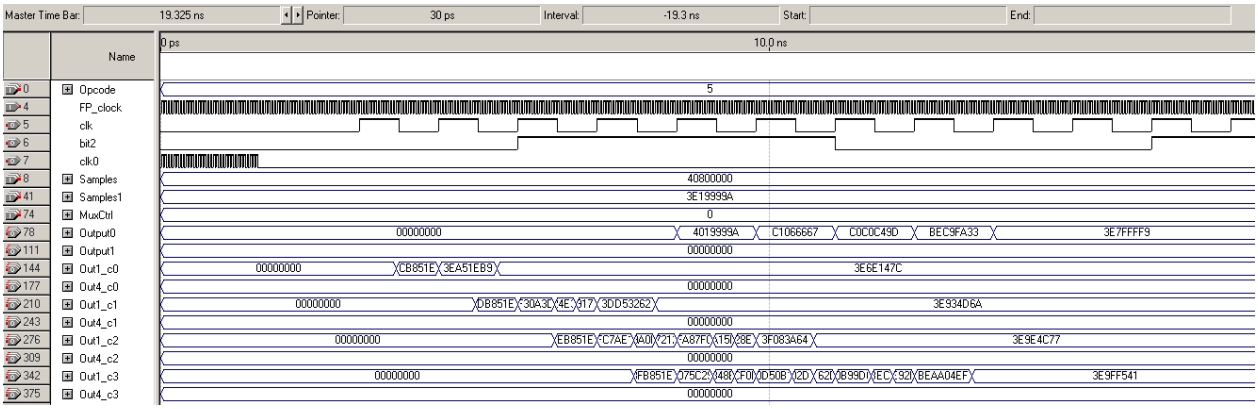


Figure 3-17 : Contrôle temporel du filtre de l'algorithme de division selon Newton-Raphson.

La précision de calcul du résultat dépend du nombre de cellules utilisées car chacune d'elles représente une itération de cet algorithme.

3.2.4 Conception sur FPGA d'une matrice cellulaire 6x4.

Dans cette section des cellules sont utilisées pour concevoir une matrice, de 6x4 UPMs, programmée et simulée pour différentes reconfigurations telles que le filtrage FIR/IIR d'ordre n=20, le filtrage FIR d'ordre n=26, le filtrage All-Zero d'ordre s=10, le filtrage All-Pole d'ordre s=10, un operateur "*butterfly*" radical 4 (OIOO) et une FFT radical 4 d'ordre N=16, le calcul de l'inverse selon Newton-Raphson, s=9 itérations et le calcul de racine carrée selon Newton-Raphson, s=9. Les entrées périphériques de ce processeur vectoriel reçoivent des données à partir des entrées de *32-bit* nommées respectivement, *Samples* et *Samples1*. Les ANNEXEs 19 à 36 montrent le programme de ce processeur et des registres à décalage qui sont utilisés pour enregistrer les données en entrée et un multiplexeur est utilisé pour contrôler les sorties des fonctions spectrales. Dans le but de permettre un grand nombre d'opérations de traitement de signaux les connexions internes entre les UPMs sont reconfigurées.

La reconfiguration des matrices cellulaires est contrôlée par un mot de *4-bit* appelé *Opcode.* Les entrées et sorties internes du processeur vectoriel sont reconnectées entre elles en changeant la valeur des bits de contrôle de la reconfiguration. Pour chaque configuration, un algorithme de connexions préprogrammées est utilisé. L'algorithme Verilog-HDL des matrices cellulaires, appelé *UPM_Arrays.v*, débute avec la déclaration de quatre horloge appelées respectivement *clk*,

clk0, bit2 (qui représente l'horloge *clk_bit2*) utilisées pour le contrôle des registres à décalage et *FP_clock* qui est l'horloge des operateurs point flottant. Une entrée de *4-bit* appelée *MuxCtrl* est employée pour le contrôle d'un multiplexeur. Les sorties de *32-bit* du processeur sont respectivement appelées *Output0* et *Output1*. Dans ce programme, afin de réduire les ressources matérielles utilisées dans le FPGA deux types d'UPMs sont utilisés, un appelé *UPMR.v* avec un contrôle de la récursivité et l'autre sans contrôle de la récursivité appelé *UPM.v* comme illustré dans la figure 3-18. Les programmes du module de récursivité, de L'UPM et de l'UPMR utilisés sont montrés dans les ANNEXEs 37 et 38. Le processeur vectoriel programmé comprend dix appels de la fonction *UPMR.v* et quatorze appels de *UPM.v*. Ces UPMs sont respectivement appelés *c0, c1, c2, ..., c23*. Dans chaque appel de la fonction UPM des extensions *_c0, _c1, _c2, ..., c_23* sont utilisées pour différencier les noms des connexions chaque UPM des autres. Des registres internes sont déclarés ainsi que les connexions internes et les paramètres. Les fonctions registres à décalage sont appelées et parmi elles une partie des registres à décalage sont connectés entre eux afin de former une entrée de donnée en pipeline et d'autres sont indépendants des autres. Un multiplexeur de 32x32-bit est appelé dans le but de contrôler les sorties des fonctions spectrales. Le contrôle temporel des vecteurs d'UPMs, inclue des algorithmes de connexions et des initialisations de paramètres pour chaque reconfiguration. À chaque montée de l'horloge temporelle *clk* le choix de la reconfiguration est contrôlé par le mot *Opcode*. Si *Opcode* est égal à la valeur de *0000b* en binaire, un vecteur de dix UPMs opèrent en corrélateur et filtrage adaptatif FIR/IIR d'ordre n=20 et le programme correspondant est montré dans les ANNEXEs 24 et 25. Si *Opcode* est égal à la valeur de *0001b* en binaire, le même vecteur est configuré en mode adaptatif FIR/IIR d'ordre n=30 et le code correspondant est programmé dans les ANNEXEs 25 et 26. Si *Opcode* est égal à la valeur de *0010b*, les cellules fonctionnent en filtre en treillis All-Pole d'ordre s=10 et l'algorithme apparait dans les ANNEXEs 26 et 27. Si *Opcode* est égal à la valeur de *0011b* en binaire, le processeur vectoriel est reconfiguré en calcul de racine carrée selon Newton-Raphson s=9 dans les ANNEXEs 27 et 28. Si *Opcode* est égal à la valeur de *0100b* en binaire, les matrices cellulaires sont reconfigurées en filtre en treillis All-Zero d'ordre s=10 et l'algorithme est montré dans l'ANNEXE 28 et 29. Si *Opcode* est égal à la valeur de *0101b* en binaire, le circuit systolique est reconfiguré en filtre en calcul de division, s=9 et cet algorithme figure dans l'ANNEXE 30. Si *Opcode* est égal à la valeur de *0110b* en binaire, la matrice d'UPMs est

reconfigurée en filtre en treillis All-Zero d'ordre *s*=10 et cet algorithme se trouve dans l'ANNEXE 31 à 36.

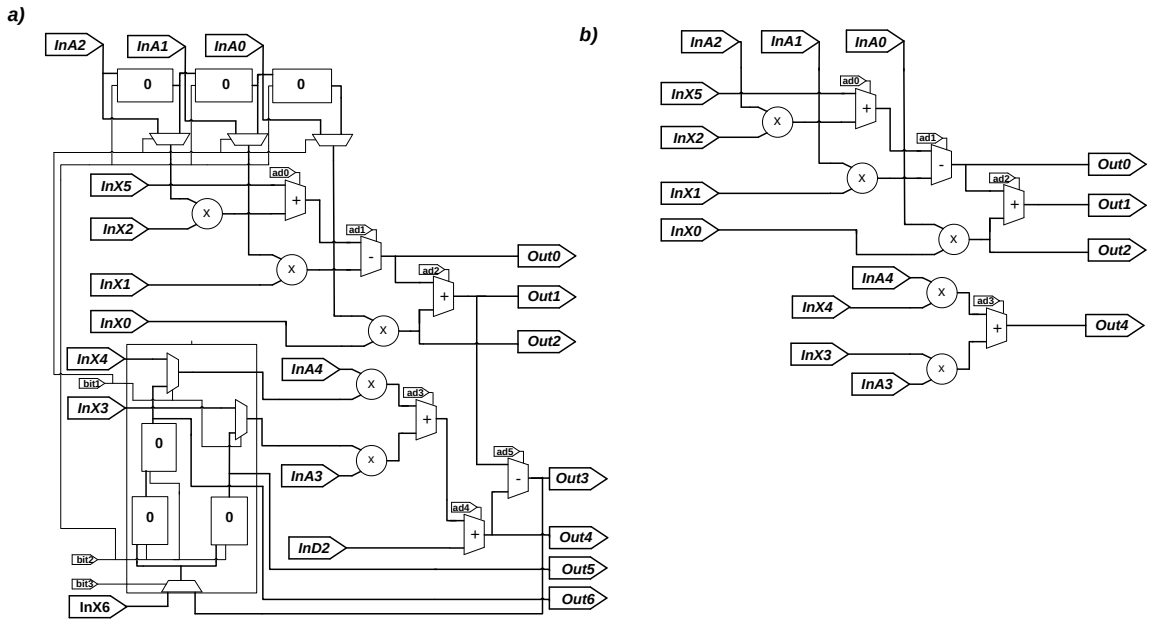


Figure 3-18 : UPM avec contrôle de récursivité *UPMR.v* a) et sans contrôle *UPM.v* b).

- **Filtrage FIR/IIR d'ordre *n*=20.**

Dans cet algorithme, un vecteur de dix *UPMRs* reçoit deux entrées appelées respectivement *Samples* et *Samples1*; ce dernier représente les valeurs des paramètres du filtre comme illustré sur les figures 3-6 a) et b). Des registres à décalage, mis en pipeline d'une profondeur de 20, sont utilisés pour décaler temporellement les valeurs du mot *Samples* à chaque montée de *clk_bit2* nommée *bit2* dans le code. L'horloge *clk0* permet d'acquérir en temps réel les valeurs des paramètres du filtre représentées par l'entrée *Samples1* afin de permettre un filtrage adaptatif et une corrélation. Des bits de contrôle sont reconfigurés en accord avec les valeurs des connexions des UPMs montrées dans les figures 2-10 à 2-14. Les paramètres du filtre sont initialisés à un pour plus de clarté et pour une vérification plus facile. La configuration de la figure 2-15 est utilisée comme motif connecté trois fois de suite pour avoir un ordre de filtrage égal à 20. Ce filtre FIR/IIR peut être non adaptatif en initialisant dans le programme des paramètres préprogrammés. Les sorties FIR et IIR sont respectivement connectées aux sorties des

processeurs appelées *Output0* et *Output1*. La figure 3-19 illustre le contrôle temporel de cette configuration. Dans cette réalisation le contrôle temporel est le même que montré pour la matrice 2x4, cependant il a été présenté dans les sections précédentes que la fréquence de l'horloge de contrôle *clk_bit2* est proportionnelle au nombre de cellules uniquement pour les fonctions *IIR*. La division de l'horloge *FP_clock* est donc multipliée proportionnellement au nombre de cellules car cet algorithme effectue une boucle fermée sur le vecteur de cellule.

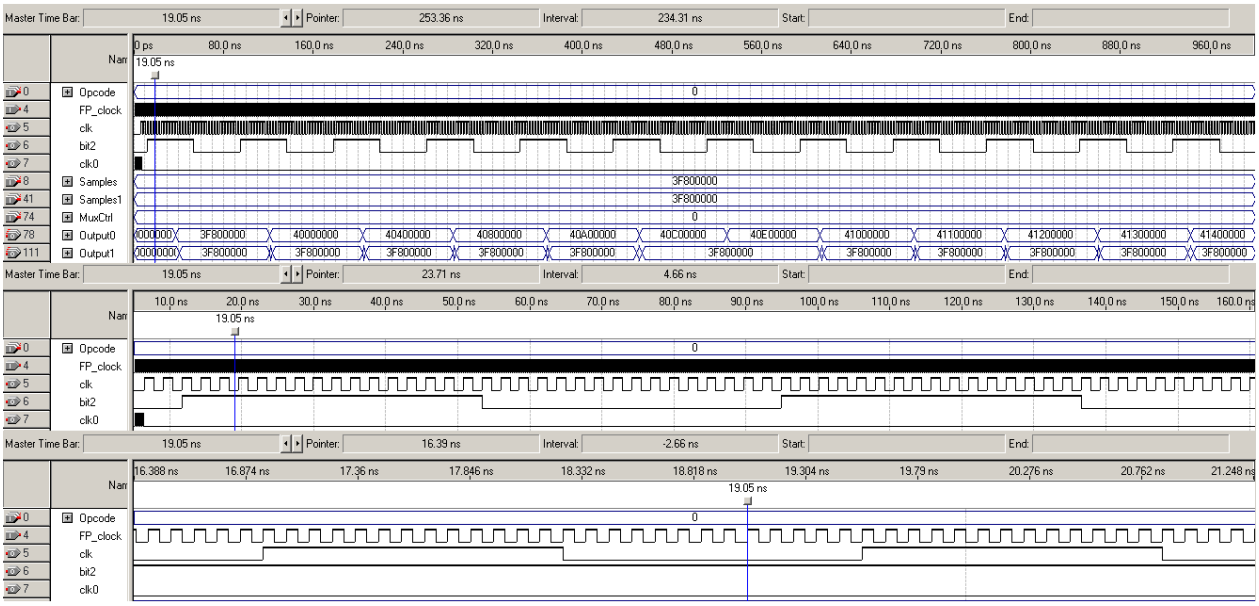


Figure 3-19 : Diagramme temporel du Filtrage FIR/IIR, ordre *N*=20.

- **Filtrage All-Pole d'ordre *s*=10.**

Dans ce filtrage *Samples* est une entrée de 32-*bit* du filtre en treillis All-Pole. La configuration des connexions du vecteur d'UPMs est illustrée à la figure 3-5 c). Des bits de contrôle sont configurés en accord avec les valeurs et connexions figurant dans 2-19 et en respect avec les noms des entrées sorties de la figure 3-20. Les paramètres du filtre sont initialisés à un pour une vérification plus facile. Les sorties du filtre All-Pole $d_{S-1}[n]$ et $\tilde{d}_s[n]$sont respectivement représentés par les sorties *Output0* et *Output1*. Le contrôle temporel de cette boucle est la même que le pour le filtrage IIR car il s'agit d'un vecteur de cellule récursif.

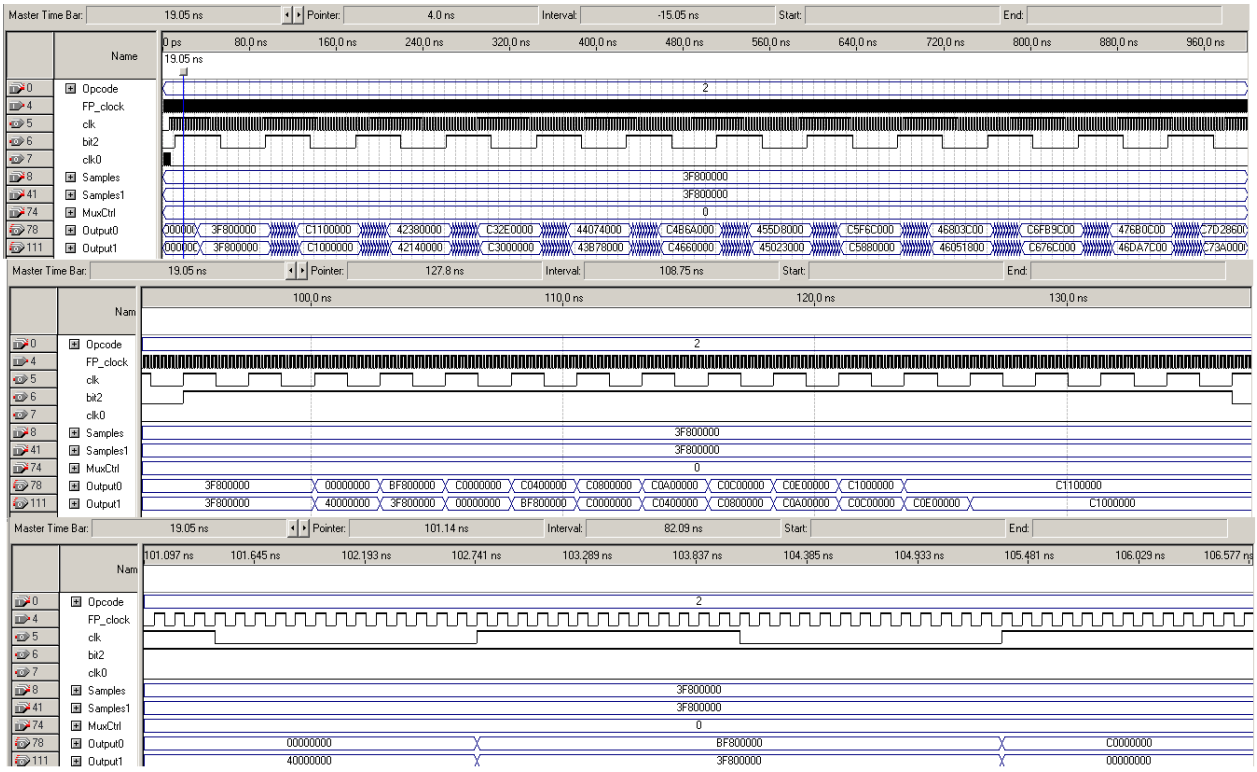


Figure 3-20 : Diagramme temporel du filtre de l'algorithme All-Pole *s*=10.

- **Filtrage All-Zero d'ordre *s*=10.**

Dans ce programme *Samples* et *Samples1* sont des entrées de 32-*bit* du filtre en treillis All-Zero dont la configuration des connexions est illustrée à la figure 3-5 d). Un registre à décalage est utilisé pour retarder la sortie. L'horloge *clk0* est synchronisée avec *clk_bit2* et contrôle le registre à décalage qui décale temporellement *Samples*[n] afin d'enregistrer *Samples*[n-1]. Des bits de contrôle sont configurés en accord avec les valeurs et connexions montrées dans la figure 2-18 et en respect avec les noms des entrées sorties de la figure 3-18. Les sorties du filtre All-Zero, nommément $e_s[n]$ et $\tilde{e}_s[n]$sont respectivement représentées par les sorties *Output0*. Ce vecteur de cellule ne fonctionne pas en boucle fermé et donc la valeur de *clk_bit2* est mise égale à celle de *clk* comme le montre la figure 3-21.

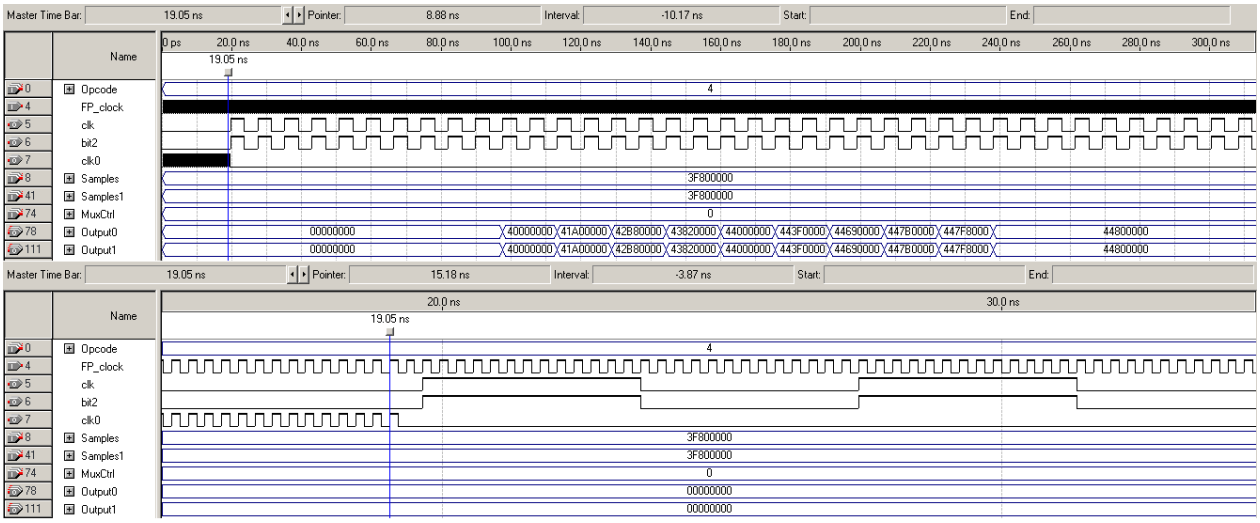


Figure 3-21 : Diagramme temporel du filtre de l'algorithme All-Zero, *s*=10.

- **Filtrage FIR *n*=30.**

Dans cette configuration, uniquement la partie supérieure de chacun des dix UPMs est utilisée dans le but d'avoir un plus grand ordre de corrélation et de filtrage FIR. La différence entre cette configuration et celle du FIR/IIR récursif vient du nombre de décalage temporel fait par les registres à décalage qui est de 30 pour cette configuration. De même que précédemment, ce vecteur de cellule ne fonctionne pas en boucle fermé et donc la valeur de *clk_bit2* est mise égale à celle de *clk* tel que le montre la figure 3-22 ou le calcul se fait en temps réel et est affiché à chaque *tap* de *clk*.

- **FFT radical 4 d'ordre *N*=16.**

Un vecteur de 24 UPMs est configuré en FFT radical 4, ordre *N*=16 et reçoit trente-deux mots en entrée (seize pour la partie réelle et seize pour la partie imaginaire). Ces données sont amenées par un vecteur de registre à décalage dont l'entrée est *Samples* comme illustrée sur la figure 3-5 e) et f). La configuration en operateur *butterfly* radical 4 illustrée sur la figure 2-28 a) est utilisée huit fois pour construire une FFT, ordre *N*=16, Fig. 2-28 b). Des bits de contrôle sont initialisés en accord avec les valeurs des bits de la configuration de la figure 2-28 a). L'operateur *butterfly* est utilisé quatre fois comme un motif pour construire la FFT. Un multiplexeur contrôlé par le

mot *MuxCltr* est utilisé pour connecter les résultats de la FFT à la sortie *Output0*. Dans le but de réduire l'utilisation des ressources du FPGA ce vecteur de *butterfly* est employé en deux itérations comme illustré à la figure 3-5 f). La première itération est programmée dans les ANNEXES 21, 24 et 25. Si le mot *clk_bit2* est égal à zéro, ce vecteur de *butterfly* calcule des valeurs intermédiaires qui seront enregistrées par des registres à décalage. Ces valeurs enregistrées sont réutilisées par le même vecteur de *butterflys* si le bit *clk_bit2* est égal à un. Dans la première itération, une partie des résultats est multipliée par des *twidle factors* qui sont déclarés comme des paramètres au début du programme. Les résultats de cette itération sont enregistrés et réutilisés durant la deuxième itération par le même vecteur pour calculer la FFT *N*=16. La figure 3-23 illustre de diagramme temporel de cette configuration.

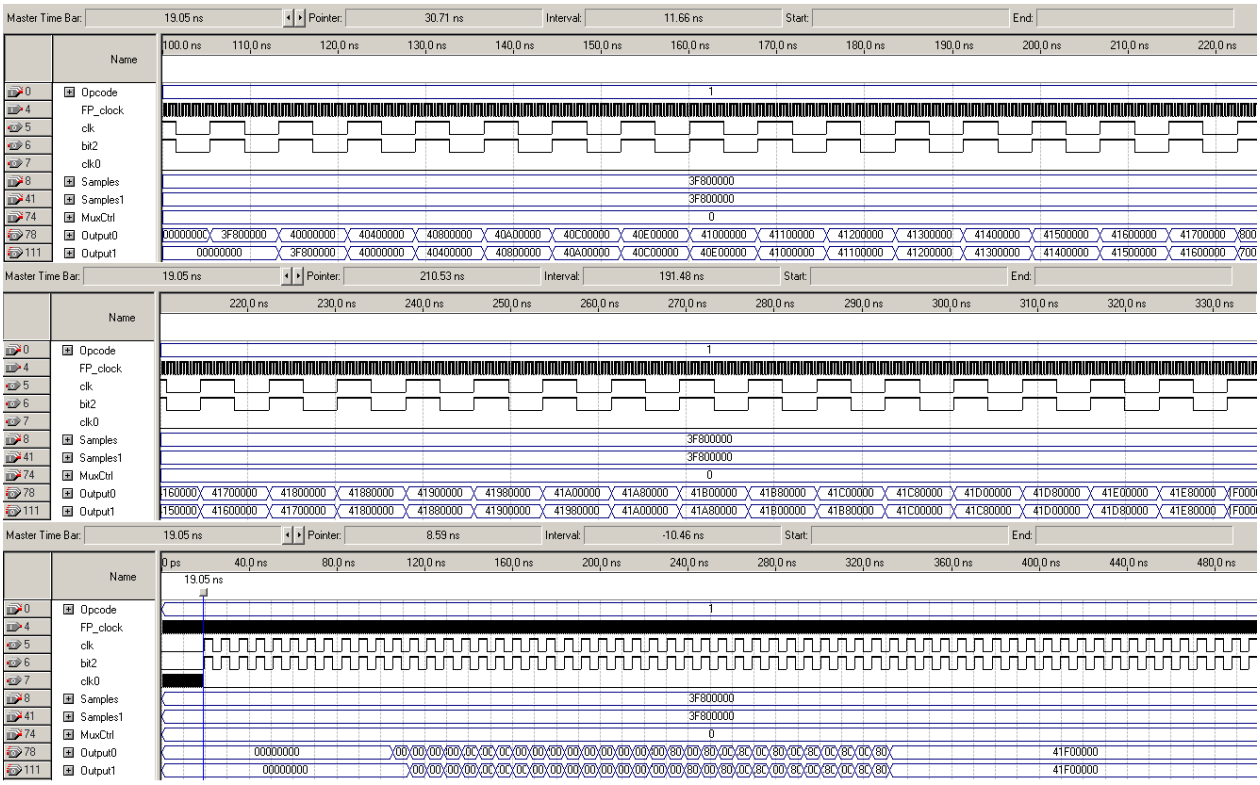


Figure 3-22 : Contrôle temporel du filtre de l'algorithme FIR, *N*=30.

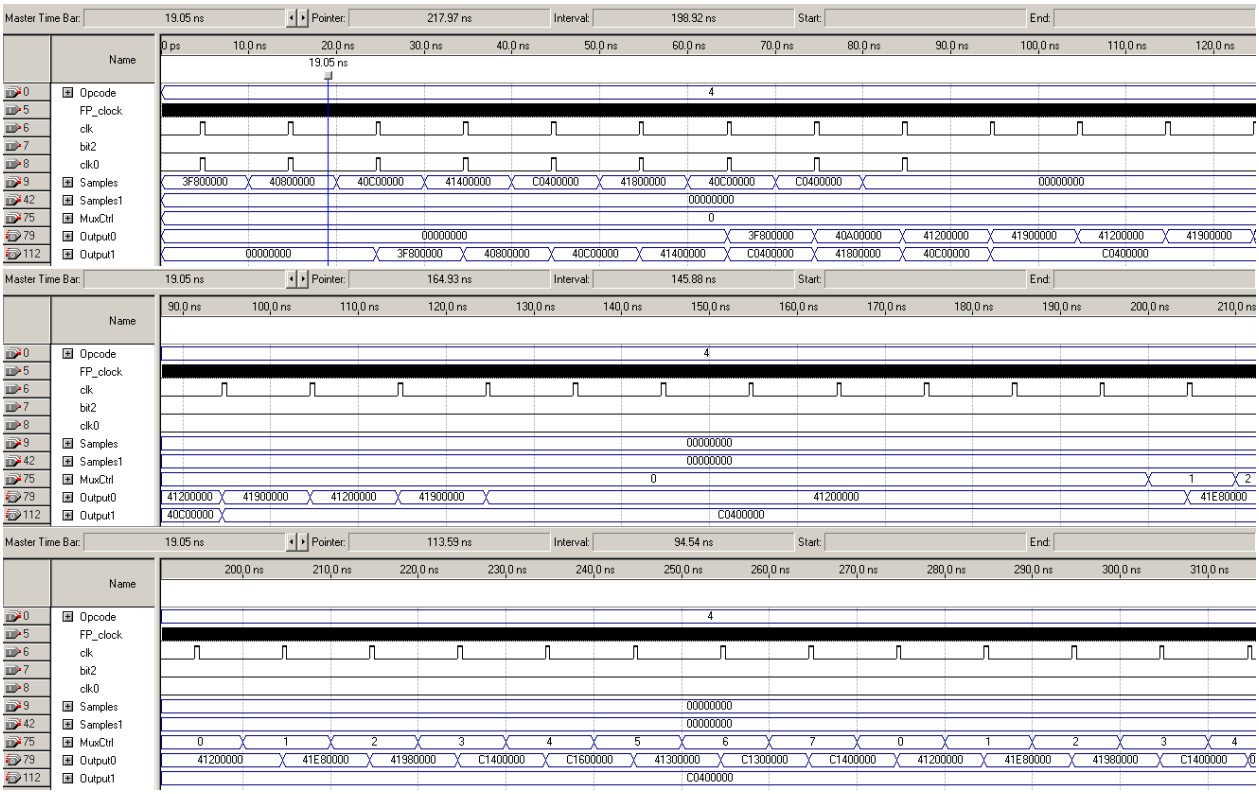


Figure 3-23 : Contrôle temporel de l'algorithme FFT radical-4, ordre *N*=16.

- **Algorithme récursif de Racine carrée selon Newton-Raphson.**

Afin de calculer une racine carrée il est nécessaire de faire un calcul d'inverse pour chaque itération de calcul de racine carrée. Comme montré dans le programme en ANNEXE 36, le port d'entrée *Samples1* est utilisé pour acquérir x_i et y_i mentionnés respectivement dans (2.78). La configuration des cellules qui effectuent la racine est la même que celle de la figure 2-34. La figure 3-24 présente un exemple de calcul de racine et la précision du calcul dépend du nombre de cellules employées car chaque cellule effectuant le calcul de racine représente une itération de cet algorithme. L'horloge *clk* est synchronisée avec *clk_bit2* car ce calcul à travers tout le processeur vectoriel ne se fait pas en boucle fermée.

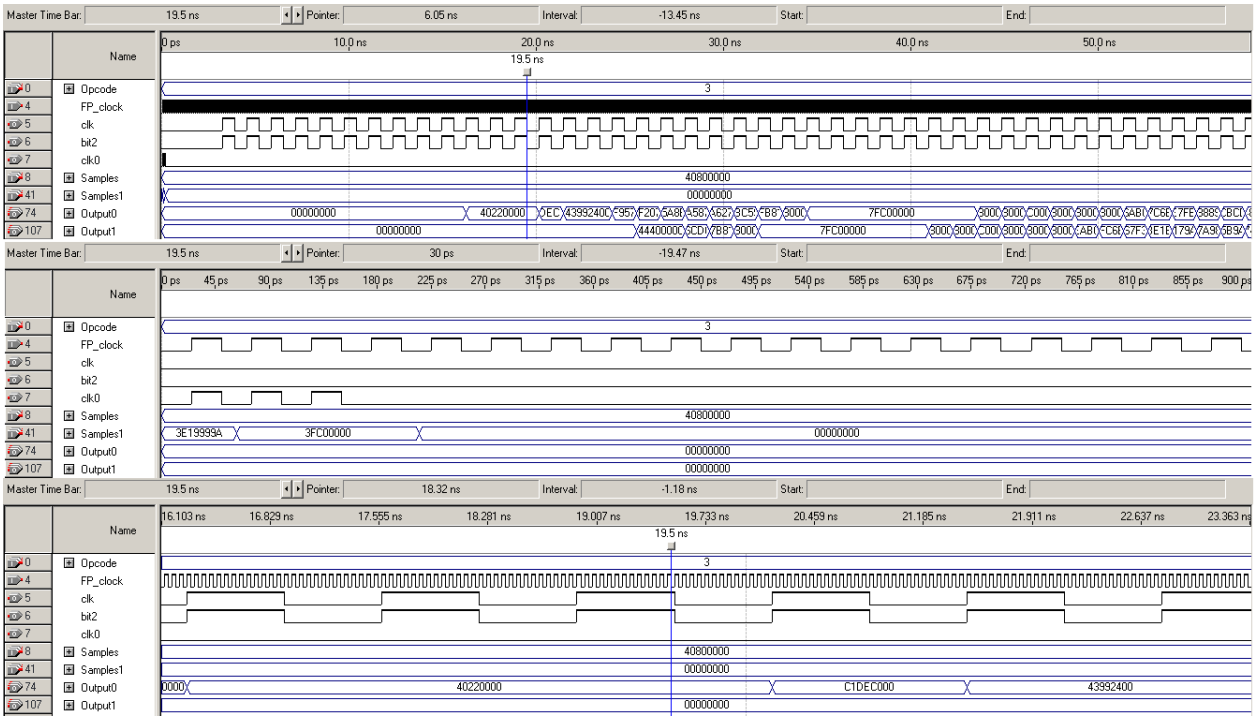


Figure 3-24 : Contrôle temporel du filtre de l'algorithme de calcul de racine selon newton-Raphson *N*=9.

3.3 Conception de matrices d'UPMs reconfigurables sur *Simulink*

Une matrice d'UPMs 2x2 a été conçue et simulée sur Matlab *Simulink* avec une configuration "*fixed-step/discrete*" avec un temps d'échantillonnage de 0.0000001 seconde. Ce modèle est illustré à la figure 3-25 et les cellules appelées *c0, c1, c2* et *c3* apparaissent à droite respectivement en vert, violet, bleue et cyan. Les ports d'entrées et sorties sont à gauche en jaune et orange. Des registres à décalage sont utilisés pour acheminer les données et une fonction sert d'algorithme de contrôle de reconfiguration des connexions tel que montré à la figure 3-25 en gris au centre et programmé à l'aide du block "*Embedded Matlab function*".

3.3.1 Contrôle des entrées et sorties de données de la matrice cellulaire.

La figure 3-26 a) illustre les entrées et sorties de la cellule qui sont en jaune. Les blocks "*Random Source*", "*Repeating Sequence Stair*" et "*Constant*" ont été initialisés à un temps d'échantillonnage de 0.0000001 seconde et utilisés pour simuler les entrées de données *Samples* et *Samples1*.

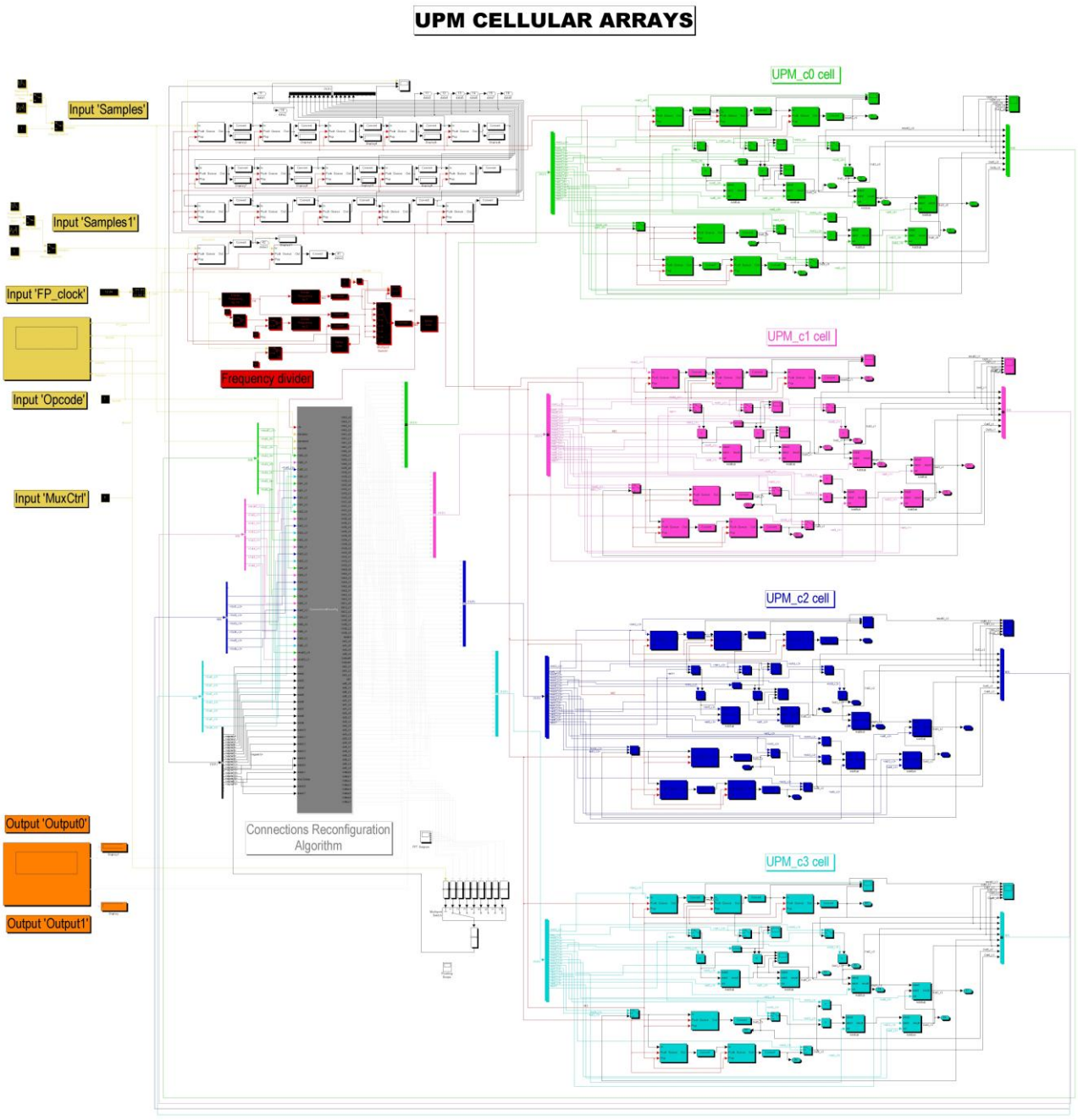


Figure 3-25 : Modèle *Simulink* d'une matrice cellulaire 2x2.

L'horloge d'entrée *FP_clock* a été simulée à l'aide du block "*Digital clock*" avec un temps d'échantillonnage de 0.0000001 seconde et le block "*pulse generator*" configuré en "*Sample based*" et en utilisant un signal extérieur. Les ports de sorties sont acheminés vers un oscilloscope par l'intermédiaire de registres illustrés sur la figure 3-27.

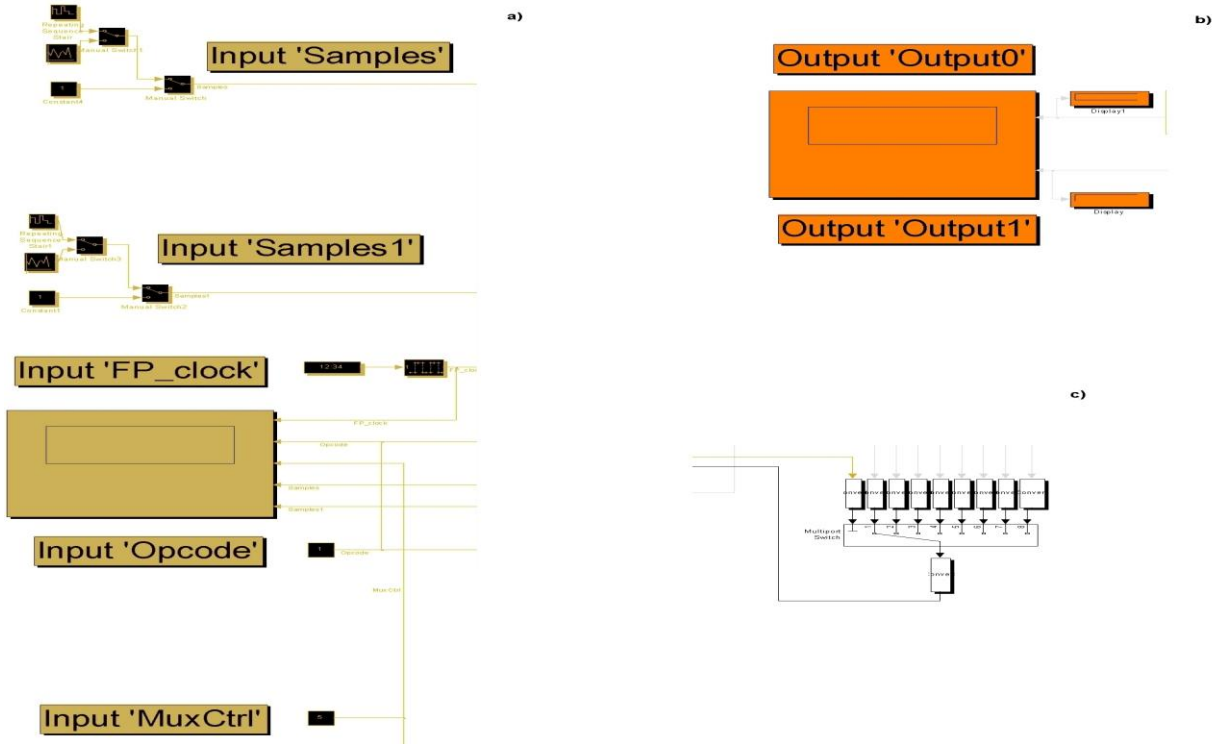


Figure 3-26 : Modèle *Simulink* des entrées et sorties du processeur.

Les entrées "*Samples*" et "*Samples1*" sont connectées à une série de blocks appelés "*Queue*" configurés en "*Rising edge*" et "*Dynamic réallocation*" et représentant des registres à décalage. Parmi ces registres certains sont connectés entre eux afin de former un pipeline de données en entrée et d'autres sont connectés indépendamment. Les données en sorties de ces registres sont envoyées vers la fonction de reconfiguration des connexions par l'intermédiaire de bus de données. Le résultat des calculs sortent de cette fonction de reconfiguration directement vers les sorties "*Output0*" et "*Output1*". Pour le calcul de fonctions nécessitant un grand nombre de sorties, un block "*Multiport switch*" représentant un multiplexeur est utilisé tel que dans 3-26 c) initialisé avec un temps d'échantillonnage de 0.0000001 seconde.

3.3.2 Contrôle temporel de la matrice.

La figure 3-28 montre le modèle *Simulink* du contrôle temporel de la matrice dans laquelle l'horloge *FP_clock* est tout d'abord divisée en *clk* par 13, par l'intermédiaire d'un block

"*Frequency divider*", ce qui donne une fréquence 26 fois égale à *FP_clock*. Le résultat *clk* de la division est elle-même divisée en 7 afin d'avoir une fréquence *clk_bit2* supérieure à 4**clk*.

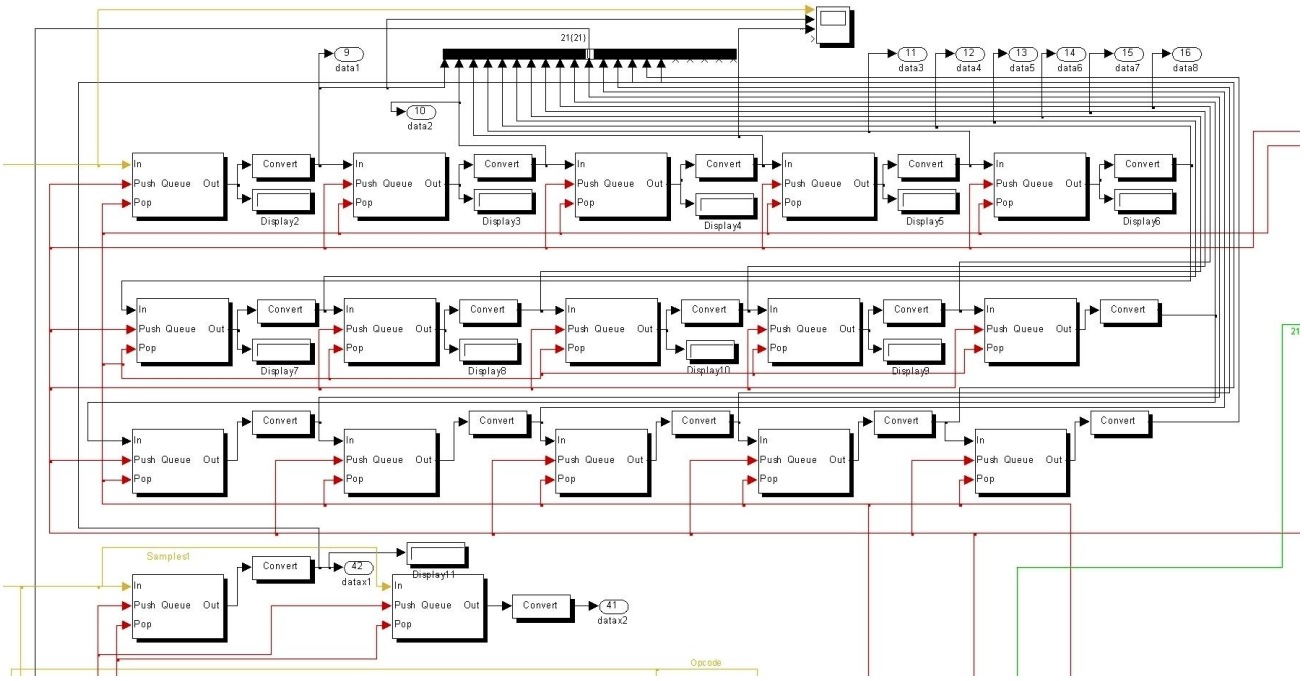


Figure 3-27 : Modèle *Simulink* d'une série de registres à décalage.

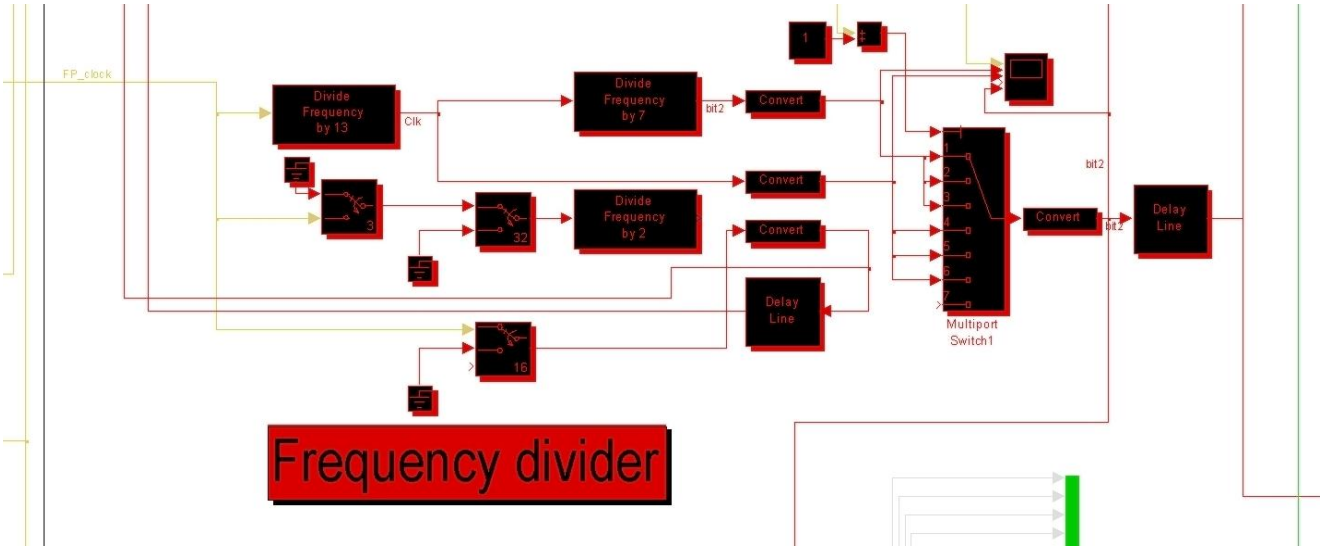


Figure 3-28 : Modèle *Simulink* du contrôle temporel de la matrice.

Des blocks "*Samples swicth*" sont utilisés pour supprimer des coups d'horloges au début et à la fin de l'acquisition de l'horloge *clk0* afin de créer des horloges destinées à l'acquisition des données. Un block "*Multiport Switch*" permet de sélectionner l'horloge utilisée en fonction de la valeur du port d'entrée *Opcode*. Un block de délai permet de retarder l'horloge en sortie du "*Multiport Switch*" afin de produite une horloge *push* qui sert comme entrée aux registres à décalage. L'horloge de sortie du "*Multiport Switch*", avant le délai, est prise comme entrée *pop* des registres à décalage. La figure 3-29 montre les horloges générées par *Simulink* à partir de l'horloge d'entrée *FP_clock*.

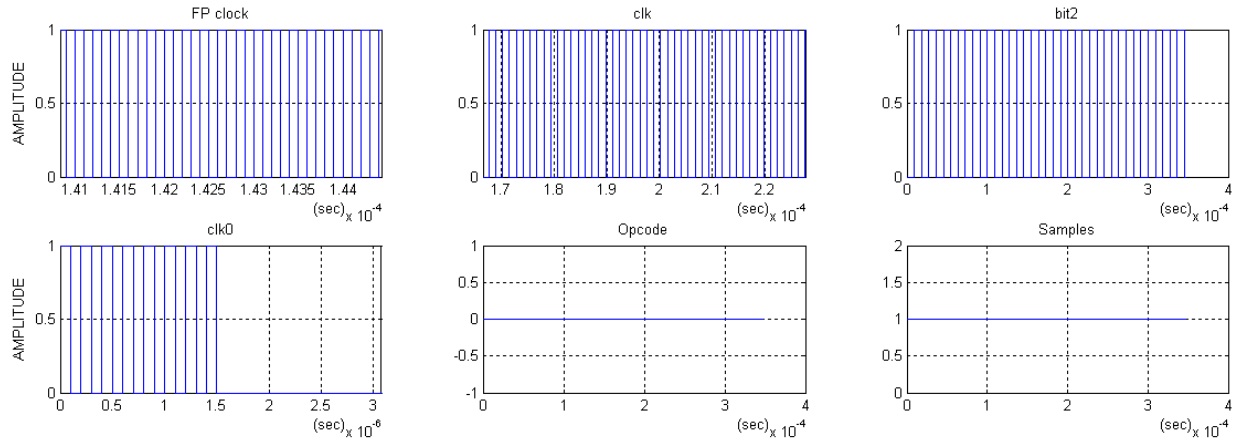


Figure 3-29 : Horloges de contrôle générées par *Simulink* à partir de l'entrée *FP_clock*.

3.3.3 Conception des cellules UPMs sur Matlab.

La figure 3-30 illustre les blocks *Simulink* utilisés pour construire l'UPM ainsi que les connexions entre ceux-ci. Des blocks "*Queue*" ont été utilisés et configurés comme précédemment en tant que registres à décalage qui sont alimentés par les horloges *push* et *pop* provenant du diviseur de fréquence. Cette cellule comprend aussi des blocks appelés "*Switch*" utilisés comme multiplexeurs entre deux données. La figure 3-31 a) montre un agrandissement

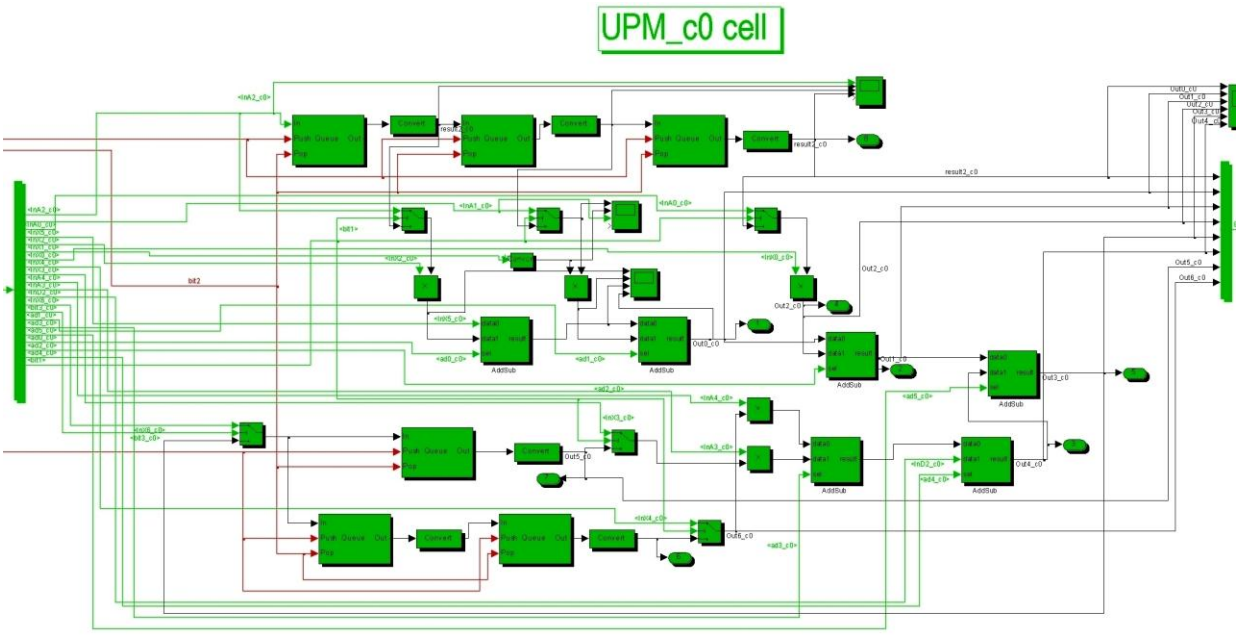


Figure 3-30 : Modèle *Simulink* d'une cellule UPM.

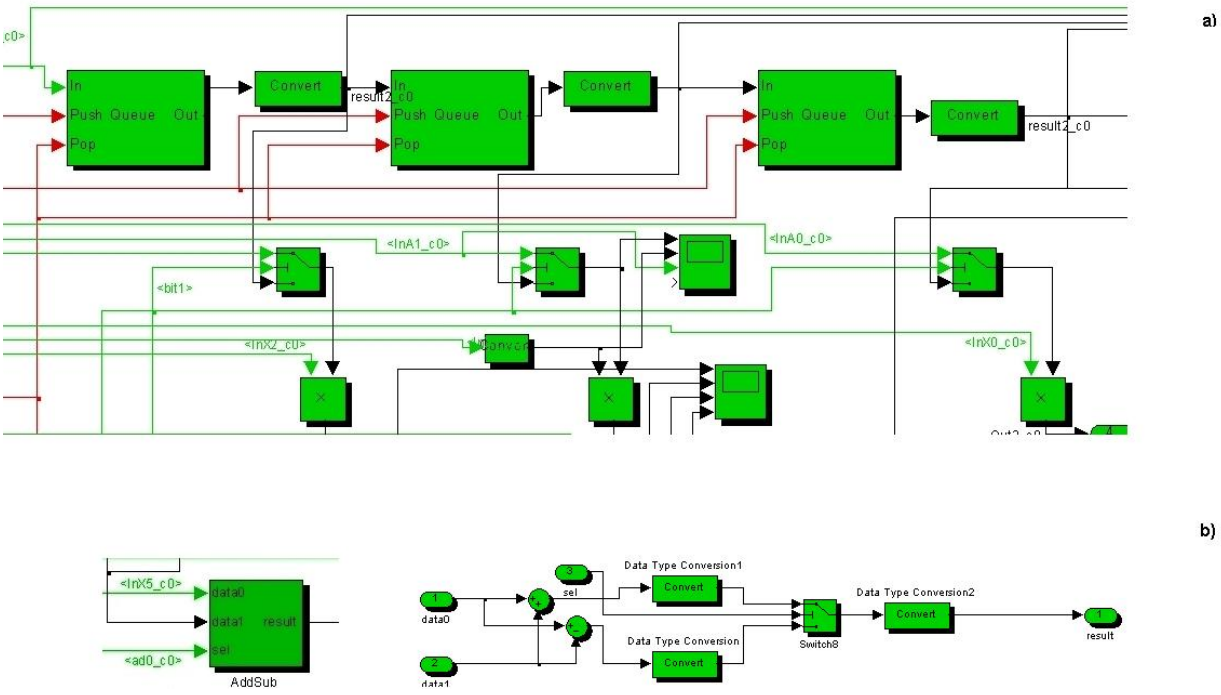


Figure 3-31 : Modèle *Simulink* d'une série de registres a) de l'additionneur/soustracteur b).

des blocks registres, "*Switch*" et multiplieur qui sont utilisés dans cette conception. Le même block "*Switch*" est utilisé avec un block additionneur pour construire une sous-fonction appelée "*Add/Sub*" qui permet de sélectionner entre une opération d'addition et de soustraction comme montré dans 3-32 b). Des connecteurs en forme de bus permettent d'acheminer les données de la fonction de reconfiguration vers chacune des cellules et d'autres permettent le transfert des données des cellules vers cette fonction de reconfiguration des connexions.

3.3.4 Contrôle de la reconfiguration des connexions entre les cellules.

La figure 3-32 présente les différentes connexions entre les bus de données et la fonction embraquée de contrôle des données simulé par le block "*Embedded Matlab Function*". Cette fonction de contrôle est montrée dans les ANNEXEs de 39 à 44. Il est à noter que lorsque le circuit fonctionne en mode adaptatif le calcul est contrôlé par données ou "*data control*" et dans ce cas cette fonction ne nécessite pas de spécification temporelle et les données sont transmises à cette fonction de l'extérieure. Par contre lorsque le circuit fonctionne en mode non adaptatif les données sont initialisées à l'intérieur de la fonction comme données continues et dans ce cas elles sont transmises de manière continue aux cellules et ceci entraine des erreurs de boucles infinies. En outre cette fonction Matlab traite les données en point fixe et ceci explique l'utilisation dans cette simulation de nombreux blocks nommés "*Convert*" qui servent à convertir les données. Tel que montré dans programme en ANNEXEs 39 à 44, lorsque le circuit fonctionne en mode adaptatif le calcul est contrôlé par donnée et dans le cas contraire, l'horloge *clk*, qui est transmise à la fonction embarquée, est utilisée comme contrôle temporel de la fonction. La déclaration de cette fonction est montrée dans l'ANNEXE 39 ainsi que l'initialisation à zéro des variables à l'intérieur de cette fonction. Dans la même ANNEXE si le mot *Opcode*=0 le processeur fonctionne en filtrage FIR/IIR non-adaptatif, ordre *N*=8 et l'algorithme de connexion est très similaire à celui utilisé pour reconfigurer la matrice 2x2 sur FPGA et correspond à la commande "*filter()*" sur Matlab. La simulation de cette configuration est affichée dans l'ANNEXE 45 montrant un grand nombre d'entrée et sortie et de connexions internes du processeur. Dans cette simulation les entrées sont configurées temporellement et les valeurs des sorties sont affichées

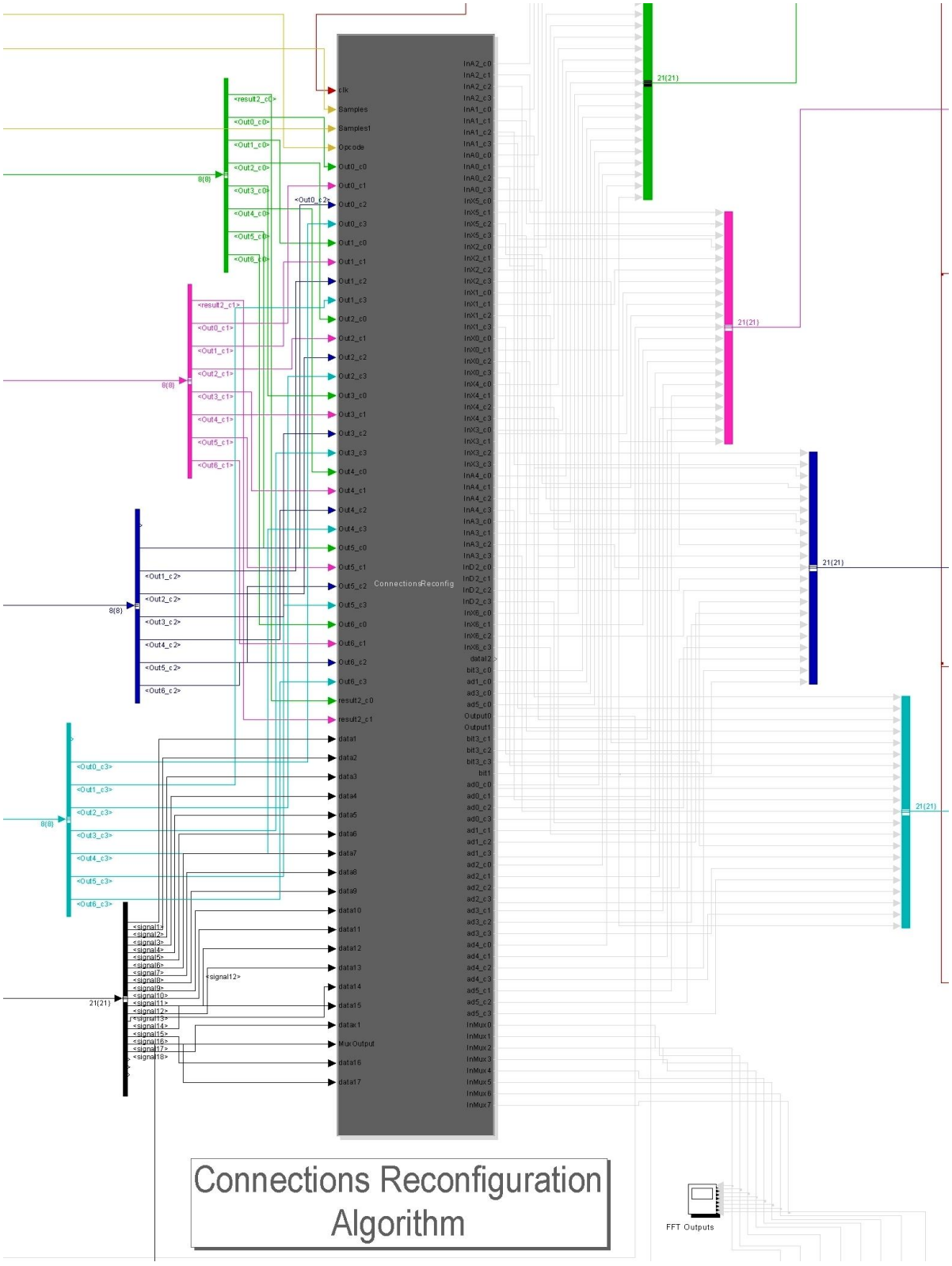


Figure 3-32 : Modèle *Simulink* de l'algorithme de reconfiguration des connexions.

sous forme de diagrammes temporels après l'exécution de la simulation. Les entrées "*Samples*" et "*Samples1*" sont mises égales à un pour de questions de simplicité. Dans l'ANNEXE 40, si le mot *Opcode*=1 le processeur fonctionne en filtrage FIR adaptatif, ordre *N*=8 et la simulation des sorties de la dernière cellule de cette configuration est montrée dans l'ANNEXE 46. Les matrices sont reconfigurées en filtrage All-Pole non adaptatif, *s*=4 étages, si *Opcode*=2 comme montré dans l'ANNEXE 41 et les résultats de cette reconfiguration est montrée dans sa simulation de l'ANNEXE 47. Les mêmes cellules fonctionnent en filtrage All-Zero, *s*=4 étages adaptatif, si *Opcode*=3, le programme et la simulation de cette configuration sont respectivement montrés dans les ANNEXEs 42 et 48. Si le mot *Opcode*=4 les cellules peuvent être utilisées en opérateur "*Butterfly*" FFT radical-4 est le programme correspondant est dans l'ANNEXE 43 et sa simulation dans l'ANNEXE 49. Un algorithme de division selon Newton-Raphson, *s*=4 étages est utilisé si *Opcode*=5 de manière adaptive comme décrit dans l'ANNEXE 44 et simulé dans l'ANNEXE 50 avec une valeur de "*Samples*" égale à 0.15 et 4 pour "*Samples1*". Un algorithme de calcul de racine selon Newton-Raphson, *s*=4 étages est utilisé si *Opcode*=6 de manière adaptive est montré dans l'ANNEXE 44 avec une simulation dans l'ANNEXE 51.

3.4 Conclusion

La conception logicielle d'un processeur reconfigurable en plusieurs fonctions différentes est réalisée dans ce chapitre. Un mot binaire est utilisé pour contrôler la reconfiguration des mêmes cellules. Un module de récursivité est aussi programmé afin d'inclure les calculs récursifs dans la listes les fonctions offertes. Une matrice de 2x2 UPMs a été programmée sur FPGA ainsi qu'une matrice 6x4 UPMs. Une simulation sur Matlab Simulink a aussi été réalisée. Un grand nombre d'algorithmes sont traités par ce processeur et parmi eux des algorithmes non récursifs, des algorithmes récursifs sur une cellule et des algorithmes récursifs sur toute la matrice de cellule. Il a été noté que la fréquence de calcul pour chaque cellule est égale à 26**FP_clock* par *tap* pour les algorithmes non récursifs. Les algorithmes récursifs sur une seule cellule ont un temps de calcul par *tap* qui est un multiple de 26**FP_ clock.* Les algorithmes récursifs, qui fonctionnent comme une boucle fermée autour de toute la matrice de cellule, ont un temps de calcul par *tap* de 28**FP_ clock* qui augmente proportionnellement avec le nombre de cellule et la valeur de l'horloge *clk_bit2* est représentée par l'équation (3.1).

CHAPITRE 4 TESTS ET PERFORMANCES DES MATRICES CELLULAIRES

Ce chapitre présente une évaluation de la vitesse de calcul pour chaque reconfiguration, une estimation de l'espace logique pris par les matrices cellulaires ainsi que les résultats des tests sur l'exactitude des algorithmes. Les résultats d'une implémentation de l'UPM sur la carte FPGA et un test sur écran VGA sont décrits. Les résultats des tests sur la précision des matrices cellulaires sont montrés ainsi que leurs certitudes respectives pour chaque algorithme. Une estimation de l'espace logique pris par le processeur est faite. Les performances de vitesse sont analysées pour différentes reconfigurations et comparées avec les performances des processeurs scalaires, des processeurs DSP et *IP block*.

4.1 Tests des algorithmes programmés

4.1.1 Tests sur écran VGA et tests des sous-fonctions

Les ANNEXEs 52 à 55 montent les résultats de l'implémentation sur FPGA et des tests sur écran VGA de l'UPM et de ses sous fonctions. Les sous-fonctions testées sont respectivement des fonctions d'addition/multiplication en point flottant, addition/multiplication complexe, opérateur *butterfly* radical 2 et FFT N=4. D'autres sous-fonctions ont été testées sur l'environnement *Quartus II* comme les fonctions registres à décalage, multiplexeur, additionneur et multiplieur point flottant, multiplication et addition complexe. Une confiance plus qu'acceptable est donnée à la précision des calculs des sous-fonctions.

4.1.2 Tests des matrices cellulaires

L'une des performances des matrices d'UPMs, par rapport à d'autres circuits systoliques, est le fait qu'elles offrent plusieurs reconfigurations des connexions entre des mêmes cellules utilisées pour le calcul de la plupart des fonctions DSP. Le fait que la fonctionnalité de ce processeur peut être modifiée en utilisant un mot binaire en tant que *Opcode* fait de celui-ci un processeur reconfigurable. Une autre fonctionnalité importante est que l'ordre de certaines fonctions peut

être augmenté par une mise en cascade de plusieurs UPMs. Les algorithmes DSP du tableau 6 ont été programmés avec une certitude, dans les tests, plus qu'acceptable. De même que l'UPM a été conçu de manière à être utilisé afin de construire des fonctionnalités plus complexes et à faire des calculs en point flottant ce qui lui donne plus de précision dans les calculs.

Il est à noter que le mode FIR/IIR adaptatif peut être utilisé pour générer des polynômes de Chebyshev montrés dans le tableau 1, qui constituent une somme cumulative. Ces mêmes polynômes sont utilisés par le mode FIR/IIR pour générer des fonctions trigonométriques, exponentielles, logarithmiques, Gamma et Bessels. Ce même mode FIR/IIR adaptatif permet de calculer la fonction de corrélation et intercorrelation, transformée de Hilbert, transformée de Hartley et de Hankel. Cependant beaucoup d'autres fonctions peuvent être rajoutées et plus particulièrement des algorithmes récursif par rapport à chaque cellule, comme ceux de Newton-Raphson et d'autres algorithmes récursif par rapport au vecteur de cellules, tel que le All-Pole et FIR/IIR, et des algorithmes d'analyse spectrale.

4.2 Performances en termes d'espace logique des matrices

Un élément logique est la plus petite unité logique que l'on retrouve dans les FPGA. L'espace logique que prend un circuit peut être défini par le nombre d'éléments logiques, le nombre de bits de mémoire et le nombre de registres que ce circuit utilise. La performance des circuits présentés ici varie en fonction du nombre d'UPMs utilisés et de la manière dont ceux-ci seront interconnectés entre eux. Chaque fois que le programme de la fonction testée est compilé, *Quartus II* affiche les ressources logiques utilisées par cette fonction, soit le nombre de bits de mémoire, le nombre d'éléments logique et le nombre de registres dans le "*Project navigator*". Les ressources logiques utilisées par l'UPM sont montrées en ANNEXE 56. De même que les ressources logiques utilisées par les matrices cellulaires de 6x4 UPMs sont présentées sur l'ANNEXE 57. Ces résultats démontrent qu'un processeur matriciel 6x4 UPMs peut être intégré dans un FPGA *Cyclone III* qui représente un FPGA de gamme moyenne cependant beaucoup de ressources logiques sont utilisées du fait de la grande quantité d'interconnexions.

Tableau 6 : Tableau montrant les tests effectués sur les algorithmes DSP et leurs certitudes.

Opcode	Tests	Certitude
Filtrage FIR/IIR Adaptatif	Par simulation sur sur *Quartus II* et sur Matlab simulink.	Très Acceptable
Filtrage FIR/IIR non- Adaptatif	Par simulation sur sur *Quartus II* et sur Matlab simulink.	
Filtrage FIR Adaptatif	Par simulation sur sur *Quartus II* et sur Matlab simulink.	
Filtrage FIR non- Adaptatif	Par simulation sur sur *Quartus II* et sur Matlab simulink.	
Filtrage All-Pole Adaptatif et non- Adaptatif	Par simulation sur sur *Quartus II* et sur Matlab simulink.	
Filtrage All-Zero Adaptatif et non- Adaptatif	Par simulation sur sur *Quartus II* et sur Matlab simulink.	
Butterfly FFT radical-2	Par simulation sur sur *Quartus II* et sur Matlab simulink.	
Butterfly FFT radical-4	Par simulation sur sur *Quartus II* et sur Matlab simulink.	
FFT *N*=16 radical-4	Par simulation sur sur *Quartus II* et sur Matlab simulink.	
Calcul de division selon Newton-Raphson	Par simulation sur sur *Quartus II* et sur Matlab simulink.	Acceptable (la précision augmente avec le nombre de cellule)
Calcul de racine selon Newton-Raphson	Par simulation sur sur *Quartus II* et sur Matlab simulink.	Acceptable (la précision augmente avec le nombre de cellule)

4.3 Performances de vitesse des matrices cellulaires

Le circuit intégré, présenté ici, a été construit avec des fonctions de multiplications et d'additions points flottants comme vu précédemment. Ces fonctions points flottants reçoivent une horloge en entrée et par conséquent la vitesse de calcul l'UPM dépend de la fréquence de cette horloge. La vitesse des unités point flottant se compte en nombre de cycles d'horloge comme spécifié dans les tests précédents et dépend du constructeur de FPGA et de la famille de ces FPGA. L'additionneur point flottant met plus de temps de calcul que le multiplieur, de ce fait la fréquence maximale de l'horloge de l'additionneur point flottant est prise comme horloge d'entrée de l'UPM [51]. L'ANNEXE 56 montre les performances théoriques d'opérateurs point flottant sur FPGAs tels que ceux offert dans le *Cyclone III*, *Stratix III*, *Stratix IV* d'*Altera* [51] qui vont de 154 à 403 *MHz*. L'environnement *TimeQuest* d'*Altera* génère une estimation de la vitesse maximale des horloges du projet qui est montrée sur le tableau 7 pour différentes conditions de

fonctionnement. En effet ce processeur est réduit en vitesse par la fréquence de l'horloge point flottant qui est de 100 *MHz*.

Les vitesses des algorithmes FIR seul, All-Zero, division et racine carrée sont montrées sur le tableau 8. Comme démontré plusieurs fois dans le précédant chapitre, le filtrage FIR seul prend à chaque *tap* un nombre égal à 26 coups d'horloge *FP_clock* fixée à 100 *MHz*. Ce qui donne une fréquence de 3.846 *MHz* par *tap* pour ces algorithmes. Dans ce mode de calcul chaque cellule représente un filtre du second ordre et à chaque *tap* le calcul d'une cellule est effectué. Cependant comme montré sur la figure 3-22, représentant la simulation du FIR *N*=30, il y a un délai initial de calcul correspondant au temps que mettent les calculs pour arriver à la sortie du processeur vectoriel. Ce délai initial est équivalent au nombre de cellule multiplié par 26 coups d'horloge *FP_clock.* En outre chaque cellule effectue 6 opérations d'additions et de multiplications donc le nombre total d'opération effectué durant toute une séquence de filtrage est de $12*N^2$. De la même manière le filtrage All-Zero offre les mêmes performances que ceux du filtrage FIR comme montré dans le tableau 8.

Sur la figure 3-17, les calculs de division et de racine carrée selon Newton-Raphson prennent un temps de calcul de 26 coups d'horloge *FP_clock* par cellule. Cependant ces algorithmes effectuent une boucle récursive autour de chaque cellule et pas autour de tout le processeur matriciel. De ce fait ils prennent au grand maximum 2*26 coups d'horloge de *FP_clock* fixée à 100 *MHz* ce qui donne une fréquence de 1.923 *MHz*. Similairement au filtrage FIR à chaque *tap* le calcul d'une itération est effectué. Il existe aussi un délai initial égal à au temps compris entre la première rentrée de données et la sortie du premier résultat. Le nombre total d'additions et de multiplications point flottant effectués par la cellule est de 5 par conséquent le nombre total d'opérations durant tout un traitement de ces algorithmes est estimé à $2.5s^2$.

Il a été observé dans le chapitre précédent que l'opérateur *butterfly* de la transformée de Fourier radical-4 prend un temps de 3 coups d'horloge de *clk* soit un temps de 3*26 coup de *FP_clock* comme démontré sur la figure 3-15. Ce qui donne une fréquence maximale de calcul de 1.282 *MHz*. De même qu'une FFT *N*=16 radical-4 comme démontré dans la figure 3-23 prends un temps de calcul de 6*26**FP_clock* coups d'horloge car la FFT *N*=16 peut être vue comme une matrice 4x2 d'opérateur *butterfly* radical 4, comme vue dans la figure 2-28, qui prend un temps de calcul de deux opérateurs *butterflys.* De la même manière une FFT ordre *N*=64 peut être vue

telle qu'une matrice de 4x2 FFT *N*=16, en termes de nombre *butterfly* radical 4, qui prend deux fois de temps de calcul de plus que celui d'un ordre *N*=16. Similairement une FFT *N*=256 peut être vue comme une matrice de 4x2 FFT *N*=64 qui prend un temps de calcul deux fois plus long. Le tableau 9 montre le temps de calcul des algorithmes FFT en assumant qu'il soit possible d'implémenter un nombre équivalent de cellules.

On appelle *biquad* un filtre linéaire récursif du second ordre. La cellule UPM peut être appelée *biquad* car elle constitue un filtre linéaire récursif du second ordre qui peut être mis en cascade. Chaque cellule effectue les calculs avec une fréquence de 3.571 *MHz* pour ce mode de fonctionnement.

Les vitesses de filtrage des matrices cellulaires sont donc réduites par la fréquence des opérateurs point flottant. En assumant que les matrices cellulaires fonctionnent en combinatoire et en assumant qu'il n'y a pas de limite du nombre de cellules, les performances des matrices cellulaires seront beaucoup plus grandes. En effet, à la différence du mode point flottant, le mode point fixe combinatoire offre moins de précision mais beaucoup plus de vitesse.

4.4 Comparaison des performances de vitesse

Dans cette section les performances en termes de vitesses des matrices cellulaires sont comparées dans un premier temps à celles des processeurs scalaires *Pentium Core 2 Duo M*, le *Pentium IV* d'*Intel* et le *NIOS II* d'*Altera*. Les matrices cellulaires sont aussi comparées aux processeurs DSP parmi les plus performants tels que les *TIGERSHARC* d'*Analog Devices* et le processeur *TMS320C6742* de *Texas Instruments*. Le module IP block contruit dans ce mémoire est aussi comparé aux performances des "IP Cores" d'*Altera* offerts dans le FPGA *Cyclone III*.

Le tableau 11 présente le temps de calcul pour différentes fonctions DSP ainsi que les horloges utilisées par ces différents processeurs qui fonctionnement tous en point flottant. La fréquence de filtrage FIR/IIR par *biquad* des matrices cellulaires est montrée égale à 3.571 *MHz* par cellule ou *biquad*, ce qui donne un temps de calcul de 280 *ns* par *biquad*. Ce qui est supérieur aux temps de calcul par *biquad* de la fonction correspondante *SP biquad filter* du processeur *TMS320C6742* [7] dont le nombre de coups d'horloge est déterminé, à partir de l'exemple donné dans [7]. Ce temps de calcul du filtre "*SP Biquad Filter*" est régie par l'équation (8 * (nombre de sorties) + 65

) / 200MHz. Pour un seul *tap*, donc pour une seule sortie, le temps de calcul est de 365 *ns* par *biquad* pour ce processeur.

Les spécifications du *SP IIR Lattice filter* donnés dans [7] montrent que pour 10 étages le nombre de coups d'horloges est de 4125. Le temps de calcul est donc de 4125 / 200*MHz*, ce qui donne un temps de calcul de 20.62 *us* pour 10 étages pour le *TMS320C6742*. Comme montré dans le tableau 11 le temps de calcul par étage des matrices cellulaires configurées en All-Pole est de 3.846/s et donc pour 10 étages les matrices auront une fréquence maximale de 0.3846*MHz*. Ce qui donne un temps de calcul de 2.6 *us* pour la reconfiguration All-pole Lattice pour 10 étages de matrices cellulaires qui offrent donc une meilleure performance.

Comme montré dans le tableau 11, les matrices cellulaires exécutent le calcul d'une FFT *N*=64 radical 4 avec une fréquence de 0.320*MHz* ce qui donne un temps de calcul de 3.125 *us*. Dans le tableau FFT section *SP Complex DIT FFT* de [7], pour *N*=64 le *TMS320C6742* met un temps de calcul de 810 coups d'horloges d'une fréquence de 200 *MHz* ce qui fait 4.05 *us* et qui est plus long que les performances des matrices cellulaires pour le même ordre. Pour *N*=256 les matrices cellulaires sont légèrement plus performantes que le *TMS320C6742* et légèrement moins performantes que les "*IP Cores*" d'*Altera* [53] tel que montré dans le tableau 11. Pour une FFT *N*=1024 les matrices cellulaires se classent deuxième en termes de performances devant les processeurs DSPs *TMS320C6742* [7] et *TIGERSHARC* [6], et " *IP Cores*" [53] qui sont les légèrement plus performants pour cette fonction. En outre cette fonction à été comparée avec les FFTs calculées par les processeurs scalaires [4,5] qui offrent des performances de vitesse très lentes par rapport aux matrices cellulaires comme montré sur le tableau 11 le *Pentium IV* prend 4.204 *ms* pour effectuer une FFT *N*=1024 et le Core 2 Duo, qui fonctionne avec une horloge de 2.4 *GHz*, prend 10 fois plus de temps que les matrices cellulaires pour faire une FFT *N*=1024. De même que le processeur scalaire NIOS II prend un temps de calcul de 0.877 *ms* en mode *software* et temps de 52.72 *us* en mode *hardware accelerated* pour calculer une FFT *N*=1024 ce qui est très long comme temps de calcul par rapport aux matrices cellulaires.

Les spécifications du *SP FIR General Purpose* sont donnés dans [7] et montrent que par *tap* le nombre de coups d'horloges est de (43 + 10 * Nr * Nh / 16), la fréquence est donc de 41.25*200*MHz* pour 3 coefficient et une seule sortie, ce qui donne un temps de calcul de 224.38 *ns* par *tap* pour le *TMS320C6742*. Comme montré dans le tableau 11 le temps de calcul par étage

des matrices cellulaires configurées en FIR et en All-zero est de 260 ns par *tap*. Cependant les vitesses des processeurs *TIGERSHARC* par *tap* sont supérieures que les matrices cellulaires.

Tableau 7 : Fréquences maximum des horloges générées par *TimeQuest* d'*Altera.*

	Slow 1200mV -40C Model Fmax Summary		**Slow 1200mV 100C Model Fmax Summary**	
Clock Name	**Fmax (*MHz*)**	**Restricted Fmax (*MHz*)**	**Fmax (*MHz*)**	**Restricted Fmax (*MHz*)**
FP_clock	100.37	100.37	88.0	88.0
Opcode	243.96	119.82	221.29	115.34
clock_divider:cl0|enable	313.58	313.58	275.41	275.41

Tableau 8 : Temps de calcul des algorithmes FIR, All-Zero, division et racine.

Fonction DSP	**Nombre de coups d'horloge par *tap***	**Fréquence maximale par *tap* (*MHz*) (performances réelles)**
Filtrage FIR	26**FP_clock*	3.846
Filtrage All-Zero	26**FP_clock*	3.846
Calcul de division selon Newton-Raphson	2*26**FP_clock*	1.923
Calcul de racine selon Newton-Raphson	2*26**FP_clock*	1.923

Tableau 9 : Temps de calcul de l'algorithme FFT pour différents ordres.

Fonction DSP	**Nombre de coups d'horloge par *tap***	**Fréquence maximale par *tap* (*MHz*)**
Butterfly FFT radical-2	3*26**FP_clock*	1.282 (performance réelle)
Butterfly FFT radical-4	3*26**FP_clock*	1.282 (performance réelle)
FFT *N*=16 radical-4	6*26**FP_clock*	0.641 (performance réelle)
FFT *N*=64 radical-4	12*26**FP_clock*	0.320 (performance estimée)
FFT *N*=256 radical-4	24*26**FP_clock*	0.16 (performance estimée)
FFT *N*=1024 radical-4	48*26**FP_clock*	0.08 (performance estimée)

Tableau 10 : Temps de calcul des algorithmes FIR/IIR et All-Pole.

Fonction DSP	**Nombre de coups d'horloge par "*Biquad*"**	**Fréquence maximale par "*Biquad*" pour une cellule (*MHz*) (performances réelles)**	**Estimation du nombre de calcul de *biquad* en fonction du nombre de cellule *N***
Filtrage FIR/IIR	28**FP_clock*	3.571	*N*
Filtrage All-Pole	28**FP_clock*	3.571	*N*

Tableau 11 : Comparaison des performances des matrices cellulaires avec d'autres processeurs.

	Matrice cellulaires	**Processeurs scalaires**			**Processeurs DSP**		**IP core**
		Core 2 Duo M	**Penti-um IV**	**Altera NIOS II**	**TIGER-SHARC**	**TMS320C6 742 Multicore**	**Altera "FFT IP core" Cyclone III**
	Point flottant	Point flottant	Point flottant	Point flottant	Point flottant	Point flottant	Point flottant
Fonction DSP	100 *MHz*	2.4 *GHz*	1.9 *GHz*	100 *MHz*	600 *MHz*	200 *MHz*	116 *MHz*
Filtrage FIR/IIR	1 / (3.846 *MHz*) = 280 *ns* par *biquad* (performance réelle)	-	-	-	-	(8 * 1 + 65) /200 *MHz* = 365 *ns* par *biquad* (performance estimée)	-
Filtrage All-Pole	1 / (3.846 *MHz*) = 280 *ns* par étage. Pour s=10, temps=2.8 *ms* (performance réelle)	-	-	-	-	4125 / 200 *MHz* = 20.625 *us* pour 10 étages (performance estimée)	-
FFT *N*=64	3.125 *us* (performance estimée)	5 *us* (performance réelle)	0.177 *ms* (performance réelle)	-	-	810 / 200 *MHz* =4.05 *us* (performance estimée)	-
FFT *N*=256 radical-4	6.25 *us* (performance estimée)	27 *us* (performance réelle)	0.907 *ms* (performance réelle)	Software 0.8776*ms* (performance réelle) Hardware accelerated 52.72*us* (performance réelle)	-	3696 / 200 *MHz* =18.48 *us* (performance estimée)	2.2 *us* (performance réelle)
FFT *N*=1024 radical-4	12.5 *us* (performance estimée)	136 *us* (performance réelle)	4.204 *ms* (performance réelle)	-	15.64 *us* (performance estimée)	9162 / 200 *MHz* =45.81*us* (performance estimée)	8.83 *us* (performance réelle)
Filtrage FIR	1 / (3.846 *MHz*) = 260 *ns* (performance réelle)	-	-	-	0.83 *ns* par *tap* (performance estimée)	(43 + 10 * 3 * 1 / 16) / 200*MHz* = 224.38 *ns* par *tap* (performance estimée)	-
Filtrage All-Zero	1 / (3.846 *MHz*) = 260 *ns* (performance réelle)	-	-	-	-	-	-

4.5 Conclusion

Dans ce chapitre les tests de plusieurs fonctions ont été effectués sur les mêmes cellules en utilisant un opcode pour contrôler la reconfiguration de ces cellules. On n'en déduit que toutes les sous-fonctions ainsi que les algorithmes DSP fonctionnent correctement avec une précision acceptable en point flottant. En outre les matrices cellulaires prennent beaucoup d'espace à cause du nombre élevé de connexions, cependant une matrice de 6x4 UPMs peut rentrer dans un FPGA *Cyclone III* dit de gamme moyenne. Les performances de vitesses ont été analysées pour chaque reconfiguration proposée et une estimation de vitesse est proposée pour chaque reconfiguration. Il est possible de conclure que les matrices cellulaires offrent de très bonnes performances pour toutes les fonctions rivalisant avec les processeurs proposés. Plus particulièrement avec les processeurs scalaires qui offrent des performances inférieures pour les FFT et le filtrage FIR. De même que les matrices rivalisent en termes de vitesse de calcul avec les processeurs DSPs les plus performants comme expliqué précédemment. En effet les résultats des matrices cellulaires pour tous les modes présentent des vitesses supérieures exceptées pour le mode FIR du *TIGERSHARC* et la FFT de l'"*IP block*" *d'Altera* qui affichent de meilleurs résultats. Les autres algorithmes testés offrent une vitesse de calcul du même ordre que les autres processeurs comparés. On remarque aussi que plus l'ordre des algorithmes calculés augmente plus les matrices cellulaires sont performantes que les autres implémentations.

CONCLUSION

Ce mémoire présente l'étude, la conception sur FPGA et les tests d'un module de propriété intellectuelle (IP Block), destiné au traitement des signaux à usage des matrices cellulaires, en point flottant reconfigurable dans le but de calculer la plupart des fonctions de DSP. Les mêmes cellules sont réutilisées par l'intermédiaire d'une reconfiguration contrôlée par un mot binaire. Les résultats des tests effectués permettent de conclure qu'il est possible de concevoir un processeur avec de telles caractéristiques. En effet après avoir introduit le sujet de recherche ce mémoire présente la réalisation théorique, la conception logicielle, les tests et l'implémentation sur FPGA des différentes configurations possibles du module de traitement universel UPM incluant le calcul de fonctions récursives.

Une revue de littérature sur les processeurs scalaires et vectoriels a été présentée, la question de recherche mentionée, ainsi que les objectifs spécifiques et les hypothèses scientifiques originales de recherche. La méthodologie employée a été décrite ainsi que les phases du projet et son échéancier. Plusieurs réalisations théoriques ont ensuite été montrées telles que les filtres FIR, IIR, multiplieur et additionneur complexe. D'autres configurations permettent à ces mêmes UPMs de fonctionner en mode filtre en treillis All-Pole, All-Zero et Pole-Zero. Des matrices de corrélations sont montrées comme des possibles implémentations en utilisant des structures cellulaires. L'architecture parallèle des matrices cellulaires proposée mène vers une évaluation rapide et efficace de la plupart des transformées telle que la transformée discrète de Fourier, discrète de Hilbert, discrète de Hartley, discrète de cosinus et la transformée discrète de Hankel. La transformée rapide de Fourier, rapide de Walsh-Hadamard, rapide généralisée de Walsh-Hadamard et d'autres analyses spectrales généralisées sont programmées en utilisant les matrices cellulaires proposées. D'autres réalisations comprennent la génération de fonctions qui emploient les polynômes de Chebyshev pour générer des fonctions trigonométriques, trigonométriques inverses, exponentielles, logarithmiques, Gamma et Bessel. La technique récursive de Newton-Raphson est utilisée pour construire des matrices cellulaires effectuant simultanément plusieurs itérations de la fonction division et racine carrée. Il a été montré que ce processeur peut aussi servir de base pour la conception de fonctions plus complexes telles que les filtrages ou des transformées en deux dimensions destinés au traitement des images.

Le logiciel *Quartus II*, le langage Verilog-HDL ont été utilisés pour la génération d'un multiplexeur de *32-bits*, des multiplieurs et additionneurs point flottant, mais aussi pour la conception du module de contrôle de la récursivité, de l'UPM. Un programme de tests par affichage VGA a été réalisé en utilisant la carte de développement *Cyclone II* d'*Altera*. Une estimation des performances en termes de fonctionnalité, d'espace logique et de vitesse de calcul a aussi été réalisée. Une matrice de 2x2 UPMs est programmée et simulée pour différentes reconfigurations tel que le filtrage FIR/IIR, ordre N=8, le filtrage All-Pole, s=4 étages, le filtrage All-Zero s=4 étages qui peuvent être utilisées de manière adaptives ou non-adaptives. Le programme contient aussi un opérateur "*Butterfly*" FFT radical-4, un algorithme de division selon Newton-Raphson s=4 étages. L'acheminement des données, des ports d'entrées vers cette matrice 2x2 est expliquée, ainsi que la sortie des résultats des cellules vers les ports de sorties du processeur. Le contrôle temporel de la matrice 2x2 est détaillé ainsi que la reconfiguration des connexions entre les cellules pour chaque mode de fonctionnement. Par la suite une seconde matrice de 6x4 est conçue en Verilog-HDL afin de tester d'autres configurations avec un plus grand nombre de cellules. Les filtrages adaptatifs FIR/IIR, ordre N=20 et FIR d'ordre N=28, sont réalisés ainsi qu'une FFT N=16 radical-4 et un algorithme de calcul de la Racine Carrée selon l'algorithme de Newton-Raphson s=4 étages qui utilise, pour chaque étage de calcul de racine, l'algorithme de division selon Newton-Raphson s=3 étages. Ce processeur à aussi été réalisé et simulé sur Matalb simulink pour les fonctions FIR/IIR, ordre N=9 et FIR d'ordre N=12, ainsi qu'une FFT radical-4, un algorithme de calcul de la Racine Carrée selon l'algorithme de Newton-Raphson s=1 et l'algorithme de division selon Newton-Raphson s=4 étages. Il a été noté qu'en mode point flottant la fréquence de calcul pour chaque cellule est égale à 26* *clk* et aussi pour les algorithmes récursifs qui fonctionnent comme une boucle fermée le temps de calcul par *tap* augmente proportionnellement avec le nombre de cellule et la valeur de l'horloge *clk_bit2* est représentée par l'équation (3.1). D'autres fonctions calculant en point flottant et qui sont récursives au niveau de chaque cellule, ne présentent pas de boucle fermée au niveau de tout le processeur vectoriel et leur temps de calcul par *tap* est égal à *clk*.

Les tests de plusieurs fonctions ont été effectués sur les mêmes cellules en utilisant un opcode pour contrôler la reconfiguration de ces cellules. On n'en déduit que toutes les sous-fonctions ainsi que les algorithmes DSP fonctionnent correctement avec une précision en point flottant. En outre une estimation de l'espace logique pris par les matrices cellulaires est donnée et celles ci

prennent beaucoup d'espace due au nombre élevé de leurs connexions, cependant une matrice de 6x4 UPMs peut rentrer dans un FPGA *Cyclone III* qui est un FPGA de gamme moyenne. Comme il a été montré dans les chapitres précédents, avec une vitesse de l'horloge point flottant de 100 *MHz* les performances de vitesse se trouvent atténuées. Les performances de vitesses ont été analysées et une estimation de vitesse est donnée pour chaque reconfiguration. Il est possible de conclure que les matrices cellulaires offrent de très bonnes performances pour toutes les fonctions rivalisant avec les processeurs proposés. Plus particulièrement avec les processeurs scalaires qui offrent des performances très inférieures pour les FFT et le filtrage FIR que les matrices proposées. De même que les matrices cellulaires rivalisent en termes de vitesse de calcul avec les processeurs DSPs les plus performants comme expliqué précédemment. En effet les résultats des matrices cellulaires pour tous les modes de reconfiguration présentent des vitesses supérieures et du même ordre de grandeur excepté pour le mode FIR du *TIGERSHARC* et la FFT de l'"*IP block*" *d'Altera* qui affichent de meilleurs résultats. On remarque aussi que plus l'ordre des algorithmes calculés augmente plus les matrices cellulaires offrent de meilleures performances que les autres implémentations. En outre l'UPM a été conçue de manière à fonctionner de manière combinatoire ce qui lui donne une vitesse de calcul plus grande.

Durant ce travail de recherche un article a été écrit, soumis pour publication et accepté à la conférence "*33rd International Conference on Telecommunications and Signal Processing - TSP 2010* ". Cette conférence s'est tenue lieu à Baden près de Vienne, en Autriche, du 17 au 20 août. La présentation de l'article, à cet événement, a été réalisée avec succès et a suscité un grand intérêt de la part des participants. Cet article a été élu parmi les meilleurs articles de la conférence [54].

L'UPM peu être vu comme un module de propriété intellectuelle (IP block) destiné à être utilisé dans un FPGA, en contraste avec l'UPE qui est un concept d'architecture pour une cellule universelle de traitement. La reconfiguration des connexions offerte par l'UPM constitue la principale différence avec l'UPE, cependant il existe bien d'autres différences au niveau de la conception théorique et de l'architecture qui sont montrés dans le tableau 12. En outre l'UPE est restée une idée qui n'a pas été implémentée.

Comme mentionné dans la revue de littérature les autres processeurs vectoriels sont très spécialisés et ont été conçus pour des applications spécifiques d'algorithmes traitement de

signaux. L'UPM propose multitude de reconfigurations qui le distinguent des autres architectures de processeurs vectoriels universels et spécialisés en plus d'offrir de grande performances de vitesses dépassant les processeurs scalaires et rivalisant avec les processeurs DSP et les *IP blocks*.

Ainsi plusieurs suggestions concernant les matrices cellulaires sont apparues dans la littérature. Leurs objectifs sont considérablement différents de ceux proposés dans ce travail qui comprend plusieurs nouveaux objectifs qui incluent la réalisation d'un grand nombre de fonctions telles que des calculs d'arithmétique complexe, d'algorithmes récursifs, des transformées, du filtrage adaptatif, parmi d'autres avec une très grande précision de calcul basée sur l'arithmétique point flottant. En outre, sa grande vitesse de calcul en point fixe et sa propriété multifonctionnelle qui est associée aux processeurs scalaires sont des avantages offerts par l'UPM. Des futurs développements dans la nanotechnologie pourraient permettre l'implémentation de milliers d'UPMs dans une seule puce, permettent l'exécution en temps réelle de centaines de fonctions DSP.

Tableau 12 : Comparaison de l'UPM avec l'UPE.

	UPE	**UPM**
Architecture	Pourrait incorporer des multiplieurs additionneurs point flottant. [10] p 47.	Comprends des additionneurs et multiplieurs point flottant ainsi que des multiplexeurs et des registres à décalage.
Fonctionnalité de l'élément de traitement	Pourrait effectuer des calculs matriciels de multiplication accumulation, multiplication, addition inversion de matrices *pxp*, FIR et IIR[10] p 47.	Effectue calculs matriciels de multiplication accumulation Effectue en plus une reconfiguration des connexions entre les même cellules afin de traiter la plupart des algorithmes.
Architecture de matrices cellulaires	Processeur contrôlé par données *data control* avec une architecture en 3D qui pourrait inclure un réseau de permutation entre les UPEs.	Processeur contrôlé par données *data control* avec une architecture en 2D et connexions entre les UPMs reconfigurables. Les reconnexions entre les cellules peuvent être reprogrammées sur FPGA.
Fonctionnalité des matrices cellulaires	Pourrait effectuer les calculs matriciels destinés à l'analyse spectrale ainsi que du filtrage adaptatif FIR et IIR.	Effectue les calculs matriciels destinés à l'analyse spectrale ainsi que des calculs de filtrage adaptatifs FIR et IIR ainsi que des calculs d'algorithmes récursifs.

BIBLIOGRAPHIE ET REFERENCES

[1] M. J. Corinthios, Signals, Systems, Transforms and Digital Signal Processing with MATLAB. *CRC Press*, Boca Raton, Fl. 2009.

[2] D. Martinez-Nieto, M. McDonnell, P. Carlston, K. Reynolds and V. Santos, "Digital Signal Processing on Intel ® Architecture," *Intel® Technology Journal*, vol. 13, no. 1, pp. 122-145, 2009.

[3] A. Buvaneswari and M. Haner, "Lattice Filter Implementation with ETSI math operations on the TMS320C62xx," *Thirty-Sixth Asilomar Conference*, vol. 2, 2002, pp. 1094-1098.

[4] R. Longbottom. "FFT Benchmark results on PCs ". [En ligne]: http://homepage.virgin.net/roy.longbottom/fftgraf%20results.htm#anchorV1S [Consulté le 1006/2011].

[5] ALTERA. "Accelerating NIOS II Systems with the C2H compiler". Version 8.0, TU-N2C2H-1.3, Aug. 2008. [En ligne] : http://www.altera.com/literature/tt/tt_nios2_c2h_accelerating_tutorial.pdf [Consulté le 10/06/2011].

[6] ANALOG DEVICES. *TIGERSHARC processor benchmarks*. [En ligne] Disponible : http://www.analog.com/en/processors-dsp/TigerSHARC/processors/TigerSHARC_benchmarks/fca.html [Consulté le 01/06/2011].

[7] TEXAS INSTRUMENTS. C674x Floating Point Benchmark. [En ligne] Disponible : http://focus.ti.com/dsp/docs/dspplatformscontento.tsp?sectionId=2&familyId=1622&tabId=2431 [Consulté le 01/06/2011].

[8] Wikipedia., ‘’Vector processor,’’ *Wikipedia*., 2010. [En ligne]. Disponible : http://en.wikipedia.org/wiki/Vector_processor. [Consulté le 17/07/2010].

[9] S.Y. Kung, "VLSl Array Processors", *IEEE ASSP Magazine*, vol. 2, pp. 4-22, 1985.

[10] M. J. Corinthios, "3D Cellular arrays for Parallel/Cascade Image/Signal Processing", Chapter 5 of *Spectral Techniques and Fault Detections*, Editor Mark Karpovsky, Academic Press, inc., 1985, pp. 217-298.

[11] F. Mayer_Linderberg, V. Beller, "An FPGA-based floating-point processor array supporting a high-precision dot product," In *Proceedings of the International IEEE Field Programmable Technology Conference*. Bangkok, (Thailand), 2006, p. 317 – 320.

[12] P. Sinha, A. Sinha, D. Basu, "A Novel Architecture of a Re-Configurable Parallel DSP Processor," In *Proceedings of the 3rd International IEEE Northeast Workshop on Circuits and Systems Conference*. Quebec City (Canada), 2005, p. 71 - 74.

[13] J. Lin, L. Lewis, and K. Lee, "A New Multi-algorithm Multichannel Cascadable Digital Filter Processor," *IEEE 1988 Custom Integrated Circuits Conference, Roshester, NY*, 1988, pp. 10.7.1-10.7.5.

[14] Kwang-Keun Lee and R.G. Deshmukh, "A High-Speed 8x8-Bit CMOS Parallel Array Processor," *IEEE Southeastcon '92*, vol. 2, 1992, pp. 828-833.

[15] K.L. Kodandapani, "Rectangular Universal Cellular Array," *Electronics Letters*, vol. 9 no.13, 1973.

[16] Y. Iwata, M. Kawamata, and T. Higuchi, "Design of Fine Grain VLSI Array Processors for Real-Time 2-D Digital Filtering," *IEEE ISCAS*, 1993, pp. 1559–1562.

[17] Y. Wan, and M.M. Fahmy, "Linear Array Processors for 2-D FIR and IIR Digital Filters," *IEEE Transactions on Circuits And Systems*, vol. 37, no. 1, pp. 1107-1110, 1990.

[18] G. Erten, and F.M. Salam, "Cellular Mixed Signal Pixel Array for Real Time Image Processing," *IEEE Conference Record of the Thirty-First Asilomar Conference on Signals, Systems & Computers*, vol. 2, 1997, pp. 1146 – 1150.

[19] U. Schmidt, and S. Mehrgardt, "Wavefront Array Processor for Video Applications," *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, 1990, pp. 307-310.

[20] J. Lee, N. Vijaykrishnan, and M.J. Irwin, "High Performance Array Processor for Video Decoding," *Proceedings of the IEEE Computer Society Annual Symposium on VLSI*, 2005, pp. 28-33.

[21] J. Miyake, M. Urano, G. Inoue, J. Yano, S. Tsubata, T. Nishiyama and S. Yamaguchi, "Architecture of 23GOPS Video Signal Processor with Programmable Systolic Array," *IEEE ISSCC Dig. Tech. Papers*, 1997, pp. 268–269.

[22] J. Gomes and V.A.N. Baroso, "Array-Based QR-LRS Multichannel Lattice Filtering," *IEEE Transactions on Signal processing*, vol. 56, no. 8, pp. 3510-3522, 2008.

[23] S. Summerfield and S. Lawson, "VLSI Implementation of Wave Digital Filters using Systolic Arrays," *IEEE International Symposium on Circuits and Systems*, vol. 2, 1990, pp. 1235-1238.

[24] D. Youn and B. Chang, "Multichannel Lattice Filter for an Adaptive Array Processor with Linear constraints," *in ICASSP'86 IEEE International conference on acoustics, speech and signal processing*, vol. 11, 1986, pp. 1829-1832.

[25] W. Hodgkiss, "The adaptive lattice array processor," *in ICASSP'81 IEEE International conference on acoustics, speech and signal processing*, vol. 6, 1981, pp. 263-266.

[26] M.J. Corinthios, "A Fast Fourier Transform for High-Speed signal Processing," *IEEE Transactions on Computers*, vol. c-20, no.8, pp. 843-846, 1971.

[27] M.J. Corinthios, K.C. Smith, J.L. Yen, "A Parallel Radix-4 Fast Fourier Transform Computer," *IEEE Transactions on Computers*, vol. c-24, no.1, pp. 80-92, 1975.

[28] P. Roberts and N. Magotra, "FFT Based VLSI Digital Array Signal Processor", *IEEE International Conference on Systems Engineering*, 1989, pp. 285-288.

[29] M. J. Corinthios, "Optimal Parallel and Pipelined Processing Through a New Class of Matrices with Application to Generalized Spectral Analysis," *IEEE Transaction on Computers*, vol. 43, no. 4, pp. 443-459, 1994.

[30] Y.A. Geadah and M.J. Corinthios, "Natural, dyadic and Sequency order algorithms and processors for the Walsh–Hadamard transform," *IEEE Transactions on Computers*, vol. c-26, no.5, 1977.

[31] L. Qi-Hu, "Signal Separation Theory using Adaptive Arrays," *in ICASSP'83 IEEE International conference on Acoustics, Speech and Signal Processing*, vol. 8, 1983, pp. 363-366.

[32] J. W. Johnson, W.M. Munden and R.W. Lawrence, "Efficiency and Performance in Coherent Detection for Synthetic Aperture Radiometry," *in IGARSS 2000 IEEE 2000 International Geoscience and Remote Sensing Symposium*, vol. 5, 2000, pp. 2266-2268.

[33] W. M. Carey, "Array Measurements and characterisations with relative velocity as parameter," *2000 MTS/IEEE Conference and Exhibition*, vol. 1, 2000, pp. 181-184.

[34] M. Marchesi, G. Orlandi and F. Piazza, "A systolic Circuit for Fast Hartley Transform," *in ISCAS'88 IEEE International Symposium on Circuits and Systems*, vol. 3, 1988, pp. 2685-2688.

[35] F. Arguello, R. Doallo and E.L. Zapata, "Semisystolic Architecture for Fast Hartley Transform: decimation in frequency and radix 2," *IEE Proceedings on Circuits, Systems and Devices*, vol. 138, no. 8, pp. 651-660, 1991.

[36] D.C. Kara and V.V. Bapeswara Rao, "A CORDIC-based Unified Systolic Architecture for Sliding Window Applications of Discrete Transform," *IEEE Transactions on Signal Processing*, vol. 44, no. 2, pp. 441-444, 1996.

[37] V.K. Anuradha and V. Visvanathan, "A CORDIC Based Programmable DXT Processor Array," *IEEE Proceedings of the Seventh International Conference on VLSI Design*, 1994, pp. 343-348.

[38] W.–H. Fang and J. –D Lee, "Efficient CORDIC-based Systolic Architecture for Discrete Hartley Transform," *IEE Proceedings on Computers and Digital Techniques*, vol. 142, no. 3, pp. 201-207, 1995.

[39] G. Jiun-In, "A New DA-Based Array for One Dimensional Discrete Hartley Transform," *in ISCAS 2001 The 2001 IEEE International Symposium on Circuits and Systems*, vol. 4, 2001, pp. 262-265.

[40] J.C. Alves, A. Puga, L. Corte-Real and J.S. Matos, "A Vector Architecture for High-Order Moments Estimation," *in ICASSP-97 IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. 5, 1997, pp. 4145-4148.

[41] F. Sayadi, M. Atri and R. Tourki, "Rapid Prototyping IP for Autocorrelation," *in ICECS 2007 The 14th IEEE International Conference on Electronics, Circuits and Systems*, 2007, pp. 298-301.

[42] M. J. Corinthios, "High Signal Processor for Vector Transformation," *U.S. Patent*, No 3, vol. 754, no. 128, pp. 1-35, Aug 20, 1973.

[43] H. Su, "Fast fourier transform performing system for e.g. signal processing application, has reconfigurable arithmetic logic unit array coupled to local memory banks, and bit reversal block to perform bit reversal on set of data points, " Hitachi ltd., *U.S. Patent*, No 2 007 260 660 A1, 05 May, 2006.

[44] J. M. Simkins, S. P. Young and J. Wong, "*Arithmetic circuit with multiplexed addend inputs*," Xilinx, Inc., *U.S. Patent*, No 7 480 690 B2, Jul 28, 2009.

[45] J. M. Simkins, S. P. Young and J. Wong, "Programmable logic device with cascading dsp slices," Xilinx, Inc., *U.S. Patent*, No 7 472 155 B2, Dec 30, 2008.

[46] A. J. Miller, "FPGA implemented bit serial multiplier and impulse response filter," Xilinx, Inc., *U.S. Patent*, No 7 567 997 B2, Jun 24, 2003.

[47] ALTERA, "Cyclone ii FPGA Development Board Reference Manual". Version 1.0.0, P25-36048-00, Oct. 2006**.**

[48] ALTERA, "Cyclone II Device Family Data Sheet", 2004**.**

[49] ALTERA. "Quartus ii Handbook", Version 9.0, QII5V1-9.0, 2009.

[50] ALTERA. "Quartus ii web edition Software", Version 11.0, Mai 2011. [En ligne] : https://www.altera.com/download/software/quartus-ii-we
[Consulté le 03/05/2011].

[51] ALTERA. "Floating-Point Megafunctions User Guide". Version 4.0, UG-01063-4.0, Jan. 2011.

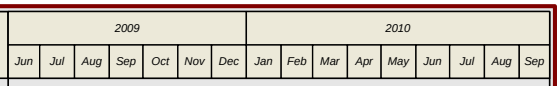
[52] Wikipedia. "Semiconductor intellectual proprety core". *Wikipedia*., 2011. [En ligne]: http://en.wikipedia.org/wiki/Semiconductor_intellectual_property_core [Consulté le 10/06/2011].

[53] ALTERA. "FFT Megacore Function User Guide". Version 11.0, UG-FFT-11.0, May. 2011. [En ligne]: http://www.altera.com/literature/ug/ug_fft.pdf [Consulté le 17/06/2011].

[54] N. El Ghali and M. J. Corinthios, "*Configurable Universal Processing Module for DSP Cellular Arrays*", Electrical Engineering Department of Ecole Polytechnique Montreal, Montreal, Canada. TSP-2010 [En ligne] : http://www.tsp.vutbr.cz/index.php?folder_id=1&file_id=27 [Consulté le 01/10/2010].

ANNEXE 1 – Diagramme de Gantt du projet.

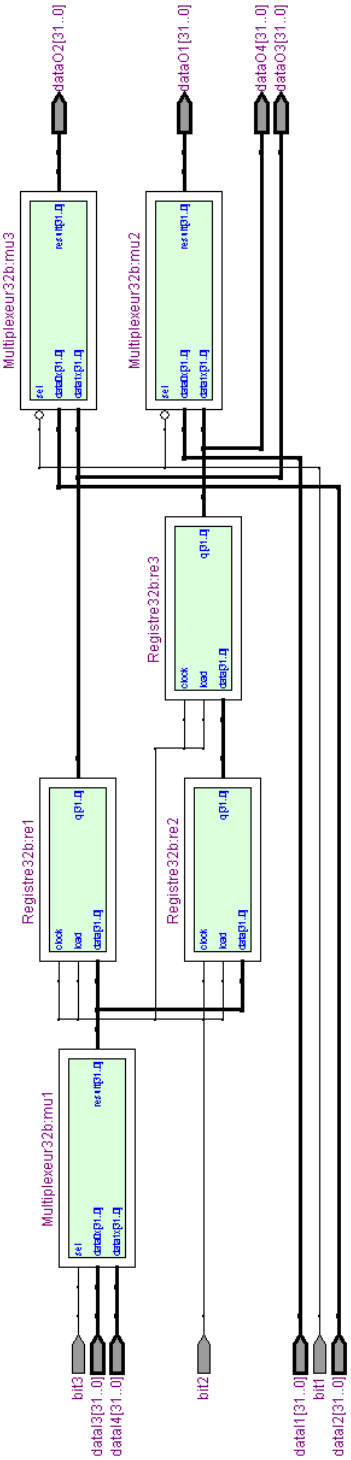
ID	Activités	Début	Fin	Durée
1	**Activités théoriques**	**27/07/2009**	**30/09/2009**	**48d**
2	Circuit FIR et IIR	27/07/2009	30/07/2009	4d
3	Circuit Multiplicateur complexe	30/07/2009	03/08/2009	3d
4	Transformée Discrète de Hankel	04/09/2009	07/09/2009	2d
5	Transformée Discrète de Hilbert	07/09/2009	08/09/2009	2d
6	Transformée Discrète de Hartley	09/09/2009	10/09/2009	2d
7	Transformée Discrète de Cosinus	11/09/2009	11/09/2009	1d
8	Filtrage en Treillis Tout-Pôles, Tout-Zéros et Zéros-Pôles	14/09/2009	16/09/2009	3d
9	Corrélations et de Cross-Corrélations	17/09/2009	18/09/2009	2d
10	Générations de fonctions trigonométriques, exponentielles, logarithmiques, Gamma el Bessel	21/09/2009	30/09/2009	8d
11	Transformée de Hankel, fonction racine carré et inverse.	30/09/2009	01/10/2009	2d
12	**Conception sur FPGA**	**14/07/2009**	**29/01/2010**	**144d**
13	Programmation de la cellule sur FPGA	01/10/2009	30/12/2009	65d
14	Programmation du circuit test	14/07/2009	29/01/2010	144d
15	Test du système et des sous-systèmes	15/10/2009	29/01/2010	77d
16	**Publication de l'article**	**07/09/2009**	**23/08/2010**	**251d**
17	Rédaction de l'article	07/09/2009	15/07/2010	224d
18	Soumission de l'article pour la conférence TSP- 2010.	28/05/2010	28/05/2010	1d
19	Soumission de la version finale de l'article TSP- 2010.	15/07/2010	15/07/2010	1d
20	Conference en Autriche	17/08/2010	23/08/2010	5d
21	**Rédaction de mémoire**	**02/12/2009**	**12/10/2010**	**225d**
22	Rédaction de mémoire	02/12/2009	12/10/2010	225d
23	Soumission du mémoire	08/10/2010	08/10/2010	1d
25	Présentation de mémoire	12/10/2010	12/10/2010	1d
27	**Valorisation**	**02/10/2009**	**12/10/2010**	**268d**
28	Déclaration d'invention	02/10/2009	15/10/2009	10d
29	Prise de contact avec Univalor	16/10/2009	16/10/2009	1d
30	Étude de brevabilité	16/10/2009	02/02/2010	78d
31	Valorisation	16/10/2009	12/10/2010	258d



ANNEXE 2 – Code Verilog-HDL du Module de Contrôle de Récursivité.

```
///////////////ControleRecurs.v//////////////
module ControleRecurs (dataI1, dataI2, dataI3, dataI4, bit1, bit2, bit3, dataO1, dataO2,
                       dataO3, dataO4);
///////Inputs data
input    [31:0]  dataI1, dataI2, dataI3, dataI4;
///////Control bits
input            bit1, bit2, bit3;
///////Output Data
output  [31:0]  dataO1, dataO2, dataO3, dataO4;
///////Internal connexions
wire    [31:0]  dataO1, dataO2, dataO3, dataO4;
wire    [31:0]  result0, result1;

////////Function calls
///////Multiplexer Function
Multiplexeur32b mu1 ( dataI3, dataI4, bit3, result0);
////////Shift register function calls
Registre32b  re1 (bit2, result0, bit2, dataO3);
Registre32b  re2 (bit2, result0, bit2, result1);
Registre32b  re3 (bit2, result1, bit2, dataO4);
//Multiplexer Function          calls
Multiplexeur32b mu2 ( dataI1, dataO4, ~bit1, dataO1);
Multiplexeur32b mu3 ( dataI2, dataO3, ~bit1, dataO2);

endmodule
```

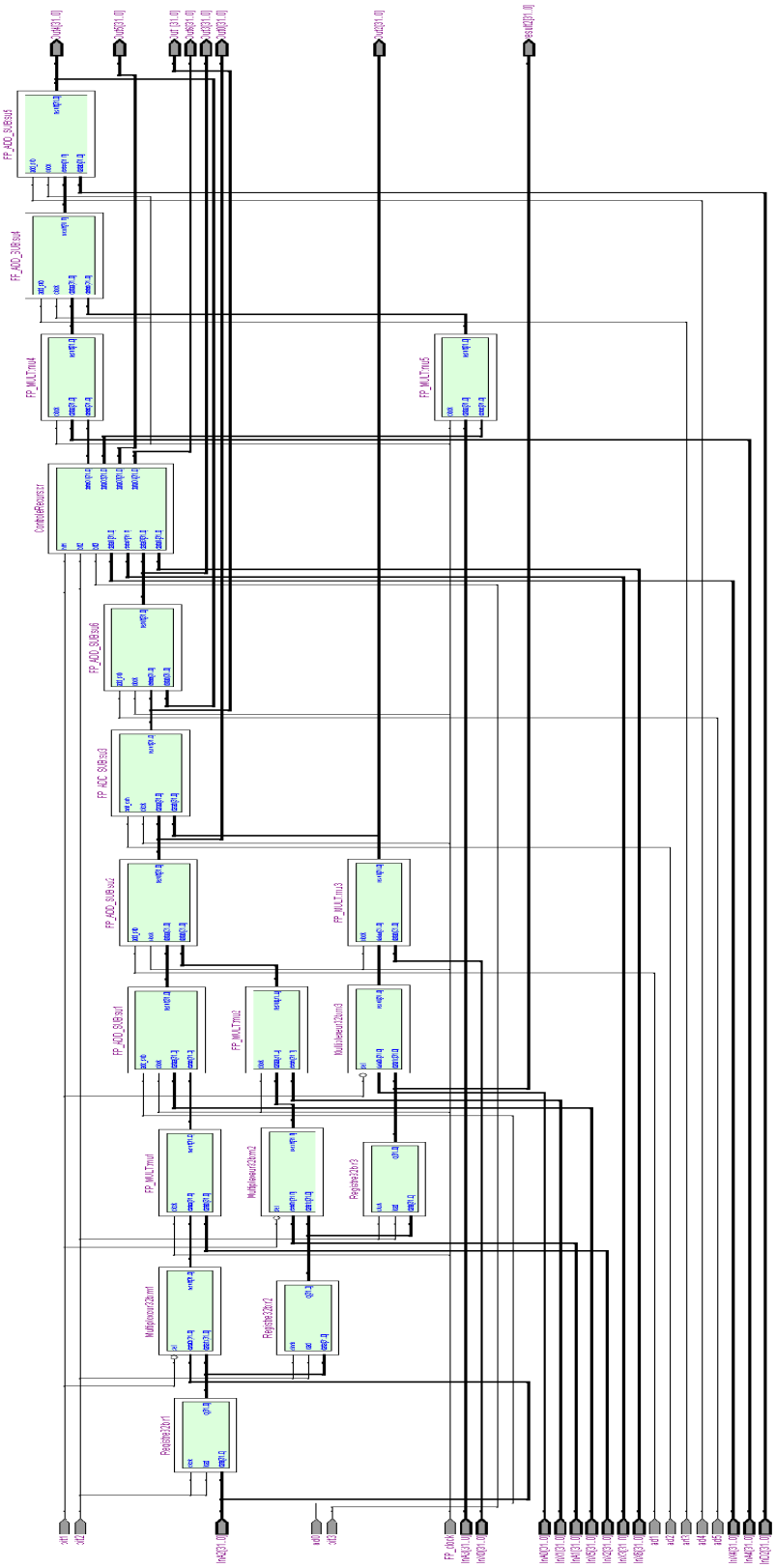
ANNEXE 3 – Diagramme interne du Module de Contrôle de Récursivité.



ANNEXE 4 – Code Verilog-HDL de L'UPM.

```
////////////////////////////////////UPM.v////////////////////////////////////////////////

////////////////Universal Processing Module instantiation
module UPM (InA2, InA1,InA0, InX5, InX2, InX1, InX0, InX4, InX3, InX6, InA4, InA3, InD2,
                                FP_clock, bit1, bit2, bit3, ad0, ad1, ad2, ad3, ad4, ad5,
        Out0, Out1, Out4, Out3, Out2, Out5, Out6, result2);  ///////Fuction instantiation
//////////////////Inputs data
input      [31:0]  InA2, InA1,InA0, InX5, InX2, InX1, InX0, InX4, InX3, InX6, InA4, InA3, InD2;
input              FP_clock, bit1, bit2, bit3, ad0, ad1, ad2, ad3, ad4, ad5;  //////Control bits
output     [31:0]    Out0, Out1, Out4, Out3, Out2, Out5, Out6, result2;   /////Output Data
wire     [31:0]  Out0, Out1, Out4, Out3, Out2, Out5, Out6;
wire     [31:0]  result0, result1, result2;          /////Internal connexions
wire     [31:0]  result3, result4, result5;          ////Internal connexions
wire     [31:0]  reg0, reg1, reg2, reg3, reg4;  ////Internal connexions
wire     [31:0]  re0, re1, re2;                      ////Internal connexions
/////Function calls
/////Recursive control call
ControleRecurs  cr (InX4, InX3, Out3, InX6, bit1, bit2, bit3, reg3, reg4, Out5, Out6);
Registre32b     r1 (bit2, InA2, bit2, result0);                    // Register function
Registre32b     r2 (bit2, result0, bit2, result1);                 /////Register function
Registre32b     r3 (bit2, result1, bit2, result2);                   //// Register function
Multiplexeur32b m1 (InA2, result0, ~bit1, result3);     /////Multiplexer Function
Multiplexeur32b m2 (InA1, result1, ~bit1, result4);     //Multiplexer Function
Multiplexeur32b m3 (InA0, result2, ~bit1, result5);     //Multiplexer Function
FP_MULT       mu1         (FP_clock, result3, InX2, reg0);   ///32-bits Floating point multiplier
FP_ADD_SUB      su1  (ad0, FP_clock, InX5, reg0, reg1);          ///32-bits Floating point adder
FP_MULT       mu2         (FP_clock, result4, InX1, reg2);   ///32-bits Floating point multiplier
FP_ADD_SUB      su2  (ad1, FP_clock, reg1, reg2, Out0);          ///32-bits Floating point adder
FP_MULT       mu3         (FP_clock, result5, InX0, Out2);   ///32-bits Floating point multiplier
FP_ADD_SUB      su3  (ad2, FP_clock, Out0, Out2, Out1);          ///32-bits Floating point adder
FP_MULT       mu4         (FP_clock, InA4, reg3, re0);       ///32-bits Floating point multiplier
FP_MULT       mu5         (FP_clock, InA3, reg4, re1);       ///32-bits Floating point multiplier
FP_ADD_SUB      su4  (ad3, FP_clock, re0, re1, re2);             ///32-bits Floating point adder
FP_ADD_SUB      su5  (ad4, FP_clock, re2, InD2, Out4);           ///32-bits Floating point adder
FP_ADD_SUB      su6  (ad5, FP_clock, Out1, Out4, Out3);      ///32-bits Floating point adder

Endmodule
```

ANNEXE 5 – Diagramme interne de L'UPM.



ANNEXE 6 – Image des outils de tests sur écran VGA.

ANNEXE 7 – Code du Module Top d'affichage VGA (1).

```
module	CII_Starter_Default
	(
		//////////////////	Clock Input	//////////////////
		CLOCK_24,		//	24 MHz
		CLOCK_27,		//	27 MHz
		CLOCK_50,		//	50 MHz
		EXT_CLOCK,		//	External Clock
		//////////////////	Push Button	//////////////////
		KEY,			//	Pushbutton[3:0]
		//////////////////	DPDT Switch	//////////////////
		SW,			//	Toggle Switch[9:0]
		//////////////////////	I2C	//////////////////////////////
		I2C_SDAT,		//	I2C Data
		I2C_SCLK,		//	I2C Clock
		//////////////////	USB JTAG link	//////////////////
		TDI,			// CPLD -> FPGA (data in)
		TCK,			// CPLD -> FPGA (clk)
		TCS,			// CPLD -> FPGA (CS)
		 TDO,			// FPGA -> CPLD (data out)

		//////////////////	VGA	//////////////////////////
		VGA_HS,			//	VGA H_SYNC
		VGA_VS,			//	VGA V_SYNC
		VGA_R,			//	VGA Red[3:0]
		VGA_G,			//	VGA Green[3:0]
		VGA_B,			//	VGA Blue[3:0]

		//////////////////	GPIO	//////////////////////////
		GPIO_0,			//	GPIO Connexion 0
		GPIO_1			//	GPIO Connexion 1
	);

//////////////////////	Clock Input	//////////////////////
input	[1:0]	CLOCK_24;		//	24 MHz
input	[1:0]	CLOCK_27;		//	27 MHz
input		CLOCK_50;		//	50 MHz
input		EXT_CLOCK;		//	External Clock
//////////////////////	Push Button	//////////////////////
input	[3:0]	KEY;			//	Pushbutton[3:0]
//////////////////////	DPDT Switch	//////////////////////
input	[9:0]	SW;			//	Toggle Switch[9:0]
//////////////////////	I2C	//////////////////////////////
inout		I2C_SDAT;		//	I2C Data
output		I2C_SCLK;		//	I2C Clock
//////////////////	USB JTAG link	//////////////////////////
input		TDI;			// CPLD -> FPGA (data in)
input		TCK;			// CPLD -> FPGA (clk)
input		TCS;			// CPLD -> FPGA (CS)
output		TDO;			// FPGA -> CPLD (data out)
//////////////////////	VGA	//////////////////////////
output		VGA_HS;			//	VGA H_SYNC
output		VGA_VS;			//	VGA V_SYNC
output	[3:0]	VGA_R;			//	VGA Red[3:0]
output	[3:0]	VGA_G;			//	VGA Green[3:0]
output	[3:0]	VGA_B;			//	VGA Blue[3:0]
//////////////////////	GPIO	//////////////////////////////
input	[35:0]	GPIO_0;		//	GPIO Connexion 0
input	[35:0]	GPIO_1;		//	GPIO Connection 1
////////////////////////////////////////////////////////////


wire		VGA_CTRL_CLK, AUD_CTRL_CLK, DLY_RST, mTCK;
wire	[9:0]	mVGA_X, mVGA_Y, mVGA_R, mVGA_G, mVGA_B, mPAR_R;
wire	[9:0]	mPAR_G, mPAR_B, mOSD_R, mOSD_G, mOSD_B, oVGA_R, oVGA_G, oVGA_B;
```

ANNEXE 8 – Code du Module Top d'affichage VGA (2).

```
wire        [19:0]  mVGA_ADDR;
reg         [27:0]  Cont;
reg                 ST;

always@(posedge CLOCK_50)       Cont    <=      Cont+1'b1;

//VGA Data 10-bit to 4-bit
assign  VGA_R       =       oVGA_R[9:6];
assign  VGA_G       =       oVGA_G[9:6];
assign  VGA_B       =       oVGA_B[9:6];
//VGA Source Select
assign  mVGA_R      =       SW[0]   ?       mPAR_R:         mOSD_R          ;
assign  mVGA_G      =       SW[0]   ?       mPAR_G          :       mOSD_G          ;
assign  mVGA_B      =       SW[0]   ?       mPAR_B:         mOSD_B          ;


VGA_Audio_PLL       u1      (       .areset(~DLY_RST), .inclk0(CLOCK_27[0]), .c0(VGA_CTRL_CLK),
                                    .c1(AUD_CTRL_CLK)
                                    );


 VGA_Controller     u2      (               //Host Side
                                    .iCursor_RGB_EN(4'h7), .oAddress(mVGA_ADDR),
                                    .oCoord_X(mVGA_X), .oCoord_Y(mVGA_Y),
                                    .iRed(mVGA_R), .iGreen(mVGA_G), .iBlue(mVGA_B),
                                    //VGA Side
                                    .oVGA_R(oVGA_R), .oVGA_G(oVGA_G),
                                    .oVGA_B(oVGA_B), .oVGA_H_SYNC(VGA_HS),
                                    .oVGA_V_SYNC(VGA_VS),
                                    //Control Signal
                                    .iCLK(VGA_CTRL_CLK), .RST_N(KEY[0]),
                                    .iRST_N(SW[0]),   .iRST_N1(SW[1]),
                                     .iRST_N2(SW[2]), .iRST_N3(SW[3])
                                    );


VGA_Pattern         u3      (               //VGA Side
                                    .oRed(mPAR_R), .oGreen(mPAR_G), .oBlue(mPAR_B),
                                    .iVGA_X(mVGA_X), .iVGA_Y(mVGA_Y),
                                    // VGA clock
                                    .iVGA_CLK(VGA_CTRL_CLK),
                                    //Control Signals
                                          .iCLK_18_4(CLOCK_50), .RST(DLY_RST),
                                    //Control Signals
                                    .RST_N(KEY[0]), .iRST_N(SW[0]),
                                     .iRST_N1(SW[1]), .iRST_N2(SW[2]), .iRST_N3(SW[3])
                                    );


endmodule
```

ANNEXE 9 – Code de la fonction d'affichage "*VGA_Pattern*" (1).

```
module   VGA_Pattern      (                    //Read Out Side
                          oRed, oGreen, oBlue, iVGA_X, iVGA_Y, iVGA_CLK,
                                               //  Control Signals
                          iCLK_18_4, RST, RST_N, iRST_N, iRST_N1, iRST_N2, iRST_N3
                          );

//Déclaration des Constantes en point flottant qui vont servir d'entrées aux fonctions
Parameter  un      =32'b00111111100000000000000000000000;  //=1
Parameter  zero    =32'b00000000000000000000000000000000;  //=0
parameter  dataa   =32'b00100101101001110010110001110010;  //=0.29e-15
parameter  datab   =32'b11101001101010000001010101000110;  //=-2.54E25
parameter  CxRa    =32'b01001110001011011100101110010100;  //=72895e4
parameter  CxIa    =32'b10111101101100011111100010100001;  //=-8.69e-2
parameter  CxRb    =32'b01000100010000011000000000000000;  //=0.774e3
parameter  CxIb    =32'b00110101000111011001001001010101;  //=587e-9
parameter  CxRc    =32'b01010011100111111010111101100001;  //=1.369e12
parameter  CxIc    =32'b01000010001010000000000000000000;  //=42
parameter  CxRd    =32'b00011101001010100100111010110011;  //=0.2254e-20
parameter  CxId    =32'b11001100110100100011000010011000;  //=-1102e5

//Déclaration des entrées sorties du VGA
output   reg       [9:0]    oRed, oGreen, oBlue;
input              [9:0]    iVGA_X, iVGA_Y;
input                       iVGA_CLK;

//Déclarations des entrées de contrôle
input                       iCLK_18_4, RST, RST_N, iRST_N, iRST_N1, iRST_N2, iRST_N3;

 //Déclarations des sorties de données de fonctions point flottant
wire               [31:0]   result_FP_Addi, result_FP_Sub, result_FP_Mult;

//Déclarations des sorties de données complexes
wire               [31:0]   result_R_CMU, result_I_CMU, result_R_CAd, result_I_CAd;
wire               [31:0]   result_Ra_But, result_Ia_But, result_Rb_But, result_Ib_But;
wire               [31:0]    result_Ra_FFT, result_Ia_FFT, result_Rb_FFT, result_Ib_FFT;
wire               [31:0]    result_Rc_FFT, result_Ic_FFT, result_Rd_FFT, result_Id_FFT;
//Registres
wire               [31:0]    reg0, reg1, reg2, reg3, reg4, reg5;  //Registres

// Appel de la fonction Addition Point Flottant
FP_ADD_SUB     FPad      ( .add_sub(1'b1), .clock(iCLK_18_4), .dataa(dataa), .datab(datab), .result(result_FP_Addi));

// Appel de la fonction Soustraction Point Flottant
FP_ADD_SUB     FPsub     (.add_sub(1'b0), .clock(iCLK_18_4), .dataa(dataa), .datab(datab), .result(result_FP_Sub) );

// Appel de la fonction Multiplication Point Flottant
FP_MULT        FPmu    ( .clock(iCLK_18_4), .dataa(dataa), .datab(datab), .result(result_FP_Mult)  );

// UPM configuré en Multiplieur Complexe
UPM            Cmu       (zero, CxIb, CxRb, CxRb, CxIb, zero, zero, zero, CxIa, CxRa, CxIa, CxRa,
                          1'b1, 1'b0, 1'b0, 1'b0, 1'b1, 1'b0, 1'b1, 1'b1, 1'b1, iCLK_18_4,
                            result_R_CMU, reg0, result_I_CMU, reg1, reg2);
```

ANNEXE 10 – Code de la fonction d'affichage "*VGA_Pattern*" (2).

```
// UPM configuré en Addition Complexe
UPM          Cad     ( zero, CxRa, CxRb, CxIa, CxIb, zero, zero, zero, un, un, un, un,
                      1'b1, 1'b0, 1'b0, 1'b0, 1'b1, 1'b1, 1'b1, 1'b1, 1'b1, iCLK_18_4,
                       result_R_CAd, reg3, result_I_CAd, reg4, reg5);

// Appel de la fonction Butterfly
Butterfly   btr0       (CxRa, CxIa, CxRb, CxIb, CxRc, CxIc,
                       iCLK_18_4, result_Ra_But, result_Ia_But, result_Rb_But, result_Ib_But);
// Appel de la fonction FFT N=4.
//FFT4      ft0         (CxRa, CxIa, CxRb, CxIb, CxRc, CxIc, CxRb, CxIb, iCLK_18_4,
                       //    result_Ra_FFT, result_Ia_FFT, result_Rb_FFT, result_Ib_FFT,
                        //   result_Rc_FFT, result_Ic_FFT, result_Rd_FFT, result_Id_FFT);

//A chaque montée de l'horloge du VGA et si le bouton poussoir n'est pas appuyé
always@(posedge iVGA_CLK or  negedge RST_N )

begin
          //Si le boutton poussoir 1 est à high
           if(!RST_N)
         begin
         end
          //Si le boutton poussoir 2 est à high
         else
         if(!iRST_N1)
         begin
         end
          //Si le boutton poussoir 3 est à high
         else
         if(!iRST_N2)
         begin
                  //Affiche en bleue les commandes suivantes
                  oBlue  <=          (iVGA_Y<10)       ?          1000:
                           //Affiche la lettre E en bleue avec une intensité de 0
                           (iVGA_Y>=10 && iVGA_Y<30 &&iVGA_X>=30 && iVGA_X<34)       ?          0:
                           (iVGA_Y>=10 && iVGA_Y<14 &&iVGA_X>=30 && iVGA_X<42)       ?          0:
                           (iVGA_Y>=18 && iVGA_Y<22 &&iVGA_X>=30 && iVGA_X<42)       ?          0:
                           (iVGA_Y>=26 && iVGA_Y<30 &&iVGA_X>=30 && iVGA_X<42)       ?          0:
                           //Affiche la lettre U en bleue avec une intensité de 0
                           (iVGA_Y>=10 && iVGA_Y<30 &&iVGA_X>=229 && iVGA_X<233)     ?          0:
                           (iVGA_Y>=26 && iVGA_Y<30 &&iVGA_X>=229 && iVGA_X<241)     ?          0:
                           (iVGA_Y>=10 && iVGA_Y<30 &&iVGA_X>=237 && iVGA_X<241)     ?          0:
                                                                                              1000;
                              ////////////////////////////////////////////////////////////////////////////////////////
                  //Affiche en rouge les commandes suivantes
                  oRed       <=          (iVGA_Y<010)                                        ?          0:
                              //Affiche la lettre E en rouge avec une intensité de 750
                             (iVGA_Y>=10 && iVGA_Y<30 &&iVGA_X>=30 && iVGA_X<34)       ?          750:
                             (iVGA_Y>=10 && iVGA_Y<14 &&iVGA_X>=30 && iVGA_X<42)       ?          750:
                             (iVGA_Y>=18 && iVGA_Y<22 &&iVGA_X>=30 && iVGA_X<42)       ?          750:
                           (iVGA_Y>=26 && iVGA_Y<30 &&iVGA_X>=30 && iVGA_X<42)         ?          750:
                                                                                              0;
                              ////////////////////////////////////////////////////////////////////////////////////////
                  //Affiche en vert les commandes suivantes
                  oGreen    <=          (iVGA_Y<10)                                         ?          0:
                              //Affiche la lettre U en vert avec une intensité de 450
                           (iVGA_Y>=10 && iVGA_Y<50 &&iVGA_X>=405 && iVGA_X<418)     ?          450:
                           (iVGA_Y>=40 && iVGA_Y<50 &&iVGA_X>=415 && iVGA_X<425)     ?          450:
```

ANNEXE 11 – Code de la fonction d'affichage "*VGA_Pattern*" (3).

```
                (iVGA_Y>=10 && iVGA_Y<50 &&iVGA_X>=422 && iVGA_X<435)      ?      450:
 ////////////////////////////////////////////////////////////////////////////////
//Affiche en vert avec une intensité de 750, de la partie imaginaire durésultat d de FFT, ordre N=4
 ////////////////////////////////////////////////////////////////////////////////
                  // Affichage de la lettre I
                (iVGA_Y>=310 && iVGA_Y<315 &&iVGA_X>=535 && iVGA_X<537) ?      750:
                 // Affichage de la lettre D
                (iVGA_Y>=312 && iVGA_Y<315 &&iVGA_X>=538 && iVGA_X<539)  ?      750:
                 (iVGA_Y>=310 && iVGA_Y<315 &&iVGA_X>=542 && iVGA_X<543) ?      750:
                 (iVGA_Y>=312 && iVGA_Y<313 &&iVGA_X>=539 && iVGA_X<542) ?      750:
                 (iVGA_Y>=314 && iVGA_Y<315 &&iVGA_X>=539 && iVGA_X<542) ?      750:
                // Affichage de la partie imaginaire durésultat d de FFT
                 (iVGA_Y>=340 && iVGA_Y<352 &&iVGA_X>=535 && iVGA_X<536)  ?      750:
                  // Affichage du bit 31 de la partie imaginaire durésultat d de FFT
 (iVGA_Y>=(350-result_Id_FFT[31]) && iVGA_Y<352 &&iVGA_X>=537 && iVGA_X<538)?   750:
                (iVGA_Y>=340 && iVGA_Y<352 &&iVGA_X>=539 && iVGA_X<540)   ?     750:
                    // Affichage du bit 30 de la partie imaginaire durésultat d de FFT
(iVGA_Y>=(350-result_Id_FFT[30]) && iVGA_Y<352 &&iVGA_X>=541 && iVGA_X<542) ?   750:
                    // Affichage du bit 29 de la partie imaginaire durésultat d de FFT
(iVGA_Y>=(350-result_Id_FFT[29]) && iVGA_Y<352 &&iVGA_X>=543 && iVGA_X<544) ?   750:
                    // Affichage des bits 28 à 26 de la partie imaginaire durésultat d de FFT
(iVGA_Y>=(350-result_Id_FFT[28]) && iVGA_Y<352 &&iVGA_X>=545 && iVGA_X<546) ?   750:
(iVGA_Y>=(350-result_Id_FFT[27]) && iVGA_Y<352 &&iVGA_X>=547 && iVGA_X<548) ?   750:
(iVGA_Y>=(350-result_Id_FFT[26]) && iVGA_Y<352 &&iVGA_X>=549 && iVGA_X<550) ?   750:
                    // Affichage du bit 25 de la partie imaginaire durésultat d de FFT
(iVGA_Y>=(350-result_Id_FFT[25]) && iVGA_Y<352 &&iVGA_X>=551 && iVGA_X<552) ?   750:
                   // Affichage du bit 24 de la partie imaginaire durésultat d de FFT
(iVGA_Y>=(350-result_Id_FFT[24]) && iVGA_Y<352 &&iVGA_X>=553 && iVGA_X<554) ?   750:
                   // Affichage des bits 23 à 0 de la partie imaginaire durésultat d de FFT
(iVGA_Y>=(350-result_Id_FFT[23]) && iVGA_Y<352 &&iVGA_X>=555 && iVGA_X<556) ?   750:
                (iVGA_Y>=340 && iVGA_Y<352 &&iVGA_X>=557 && iVGA_X<558)   ?     750:
(iVGA_Y>=(350-result_Id_FFT[22]) && iVGA_Y<352 &&iVGA_X>=559 && iVGA_X<560) ?   750:
(iVGA_Y>=(350-result_Id_FFT[21]) && iVGA_Y<352 &&iVGA_X>=561 && iVGA_X<562) ?   750:
(iVGA_Y>=(350-result_Id_FFT[20]) && iVGA_Y<352 &&iVGA_X>=563 && iVGA_X<564) ?   750:
(iVGA_Y>=(350-result_Id_FFT[19]) && iVGA_Y<352 &&iVGA_X>=565 && iVGA_X<566) ?   750:
(iVGA_Y>=(350-result_Id_FFT[18]) && iVGA_Y<352 &&iVGA_X>=567 && iVGA_X<568) ?   750:
(iVGA_Y>=(350-result_Id_FFT[17]) && iVGA_Y<352 &&iVGA_X>=569 && iVGA_X<570) ?   750:
(iVGA_Y>=(350-result_Id_FFT[16]) && iVGA_Y<352 &&iVGA_X>=571 && iVGA_X<572) ?   750:
(iVGA_Y>=(350-result_Id_FFT[15]) && iVGA_Y<352 &&iVGA_X>=573 && iVGA_X<574) ?   750:
(iVGA_Y>=(350-result_Id_FFT[14]) && iVGA_Y<352 &&iVGA_X>=575 && iVGA_X<576) ?   750:
(iVGA_Y>=(350-result_Id_FFT[13]) && iVGA_Y<352 &&iVGA_X>=577 && iVGA_X<578) ?   750:
(iVGA_Y>=(350-result_Id_FFT[12]) && iVGA_Y<352 &&iVGA_X>=579 && iVGA_X<580) ?   750:
(iVGA_Y>=(350-result_Id_FFT[11]) && iVGA_Y<352 &&iVGA_X>=581 && iVGA_X<582) ?   750:
(iVGA_Y>=(350-result_Id_FFT[10]) && iVGA_Y<352 &&iVGA_X>=583 && iVGA_X<584) ?   750:
(iVGA_Y>=(350-result_Id_FFT[9]) && iVGA_Y<352 &&iVGA_X>=585 && iVGA_X<586)   ?   750:
(iVGA_Y>=(350-result_Id_FFT[8]) && iVGA_Y<352 &&iVGA_X>=587 && iVGA_X<588)   ?   750:
(iVGA_Y>=(350-result_Id_FFT[7]) && iVGA_Y<352 &&iVGA_X>=589 && iVGA_X<590)   ?   750:
(iVGA_Y>=(350-result_Id_FFT[6]) && iVGA_Y<352 &&iVGA_X>=591 && iVGA_X<592)   ?   750:
(iVGA_Y>=(350-result_Id_FFT[5]) && iVGA_Y<352 &&iVGA_X>=593 && iVGA_X<594)   ?   750:
(iVGA_Y>=(350-result_Id_FFT[4]) && iVGA_Y<352 &&iVGA_X>=595 && iVGA_X<596)   ?   750:
(iVGA_Y>=(350-result_Id_FFT[3]) && iVGA_Y<352 &&iVGA_X>=597 && iVGA_X<598)   ?   750:
(iVGA_Y>=(350-result_Id_FFT[2]) && iVGA_Y<352 &&iVGA_X>=599 && iVGA_X<600)   ?   750:
(iVGA_Y>=(350-result_Id_FFT[1]) && iVGA_Y<352 &&iVGA_X>=601 && iVGA_X<602)   ?   750:
(iVGA_Y>=(350-result_Id_FFT[0]) && iVGA_Y<352 &&iVGA_X>=603 && iVGA_X<604)   ?   750:
(iVGA_Y>=340 && iVGA_Y<352 &&iVGA_X>=605 && iVGA_X<606)   ?      750:
                                                                                 0;
        end
end
endmodule
```

ANNEXE 12 – Image de l'affichage des tests.

ANNEXE 13 – Algorithme du processeur matriciel *2x2* UPMs (1)

```
module UPM_ARRAYS ( FP_clock, Opcode, MuxCtrl, Samples, Samples1, Output0, Output1);
//Déclaration des ports d'entrées et sorties
input               FP_clock; //Horloge des opérateurs points flottant
input       [31:0]  Samples, Samples1;   //Entrée du processeur matriciel
input   [2:0]           Opcode;                                 //Contôle la reconfiguration
input   [2:0]   MuxCtrl;            //Controle de multiplexeur
output      [31:0]      Output0, Output1;  //  Sories du processeur matriciel
reg     [31:0]  Output0, Output1;           //Sories du processeur matriciel


//Registres Internes du processeur
reg     [31:0]  InA2_c0, InA2_c1, InA2_c2, InA2_c3;
reg     [31:0]  InA1_c0, InA1_c1, InA1_c2, InA1_c3;
reg     [31:0]  InA0_c0, InA0_c1, InA0_c2, InA0_c3;
reg     [31:0]  InX5_c0, InX5_c1, InX5_c2, InX5_c3;
reg     [31:0]  InX2_c0, InX2_c1, InX2_c2, InX2_c3;
reg     [31:0]  InX1_c0, InX1_c1, InX1_c2, InX1_c3;
reg     [31:0]  InX0_c0, InX0_c1, InX0_c2, InX0_c3;
reg     [31:0]  InX4_c0, InX4_c1, InX4_c2, InX4_c3;
reg     [31:0]  InX3_c0, InX3_c1, InX3_c2, InX3_c3;
reg     [31:0]  InA4_c0, InA4_c1, InA4_c2, InA4_c3;
reg     [31:0]  InA3_c0, InA3_c1, InA3_c2, InA3_c3;
reg     [31:0]  InD2_c0, InD2_c1, InD2_c2, InD2_c3;
reg     [31:0]  InX6_c0, InX6_c1, InX6_c2, InX6_c3;
reg     [31:0]  dataI1, dataI2, dataI3, dataI4, dataI5, dataI6, dataI7, dataI8;
reg             bit3_c0, bit3_c1, bit3_c2, bit3_c3, bit1, ad0_c0, ad0_c1, ad0_c2, ad0_c3;
reg             ad1_c0, ad1_c1, ad1_c2, ad1_c3, ad2_c0, ad2_c1, ad2_c2, ad2_c3;
reg             ad3_c0, ad3_c1, ad3_c2, ad3_c3, ad4_c0, ad4_c1, ad4_c2, ad4_c3;
reg             ad5_c0, ad5_c1, ad5_c2, ad5_c3;
reg     [31:0]  InMux0, InMux1, InMux2, InMux3;
reg     [31:0]  InMux4, InMux5, InMux6, InMux7;
//Connections Internes du processeur
wire    [31:0]  Out0_c0, Out0_c1, Out0_c2, Out0_c3;
wire    [31:0]  Out1_c0, Out1_c1, Out1_c2, Out1_c3;
wire    [31:0]  Out2_c0, Out2_c1, Out2_c2, Out2_c3;
wire    [31:0]  Out3_c0, Out3_c1, Out3_c2, Out3_c3;
wire    [31:0]  Out4_c0, Out4_c1, Out4_c2, Out4_c3;
wire    [31:0]  Out5_c0, Out5_c1, Out5_c2, Out5_c3;
wire    [31:0]  Out6_c0, Out6_c1, Out6_c2, Out6_c3;
wire    [31:0]  result2_c0, result2_c1, result2_c2, result2_c3;
wire    [31:0]  data1, data2, data3, data4, data5, data6, data7, data8;
wire    [31:0]  data9, data10, data11, data12, data13, data14, data15;
wire    [31:0]  datax1, datax2, datax3, datax4, datax5, datax6, datax7, datax8;
wire        [31:0]      MuxOutput, data16, data17;
wire                        clk, clk0, bit2;
wire            S0, S1, S2, clk_Opcode0, bit2_Opcode0, clk0_Opcode0, clk0_Opcode1;
//Initialisation des paramètres
parameter           un          =       32'b00111111100000000000000000000000;  //=1
parameter           moinsun  =          32'b10111111100000000000000000000000;  //=-1
parameter           zero        =       32'b00000000000000000000000000000000;  //=0
parameter           two         =       32'b01000000000000000000000000000000;  //=2
parameter           moinstwo=       32'b11000000000000000000000000000000;  //=-2
parameter   zero_p_cinq=    32'b00111111000000000000000000000000;  //0.5
parameter   RW1     =       32'b00111111011011001000010010110110;  //=0.9239 - 0.3827i
parameter   IW1     =       32'b10111110110000011111000101000001;
parameter   RW2     =       32'b00111111001101010000010010000001;  //0.7071 - 0.7071i
parameter   IW2     =       32'b10111111001101010000010010000001;
parameter   RW3     =       32'b00111110110000011111000101000001;  //0.3827 - 0.9239i
parameter   IW3     =       32'b10111111011011001000010010110110;
```

ANNEXE 14 – Algorithme du processeur matriciel *2*x*2* UPMs (2)

```
parameter   RW6    =      32'b10111111001101010000010010000001;   //-0.7071 - 0.7071i
parameter   IW6    =     32'b10111111001101010000010010000001;
parameter   RW9    =      32'b10111111011011001000010010110110;   //-0.9239 + 0.3827i
parameter   IW9    =     32'b00111110110000111111000101000001;

//Appels de fonctions UPMs
UPM  c0 (InA2_c0, InA1_c0,InA0_c0, InX5_c0, InX2_c0, InX1_c0, InX0_c0, InX4_c0, InX3_c0, InX6_c0, InA4_c0,
        InA3_c0, InD2_c0, FP_clock, bit1, bit2, bit3_c0, ad0_c0, ad1_c0, ad2_c0, ad3_c0, ad4_c0,
        ad5_c0, Out0_c0, Out1_c0, Out4_c0, Out3_c0, Out2_c0, Out5_c0, Out6_c0, result2_c0);  //UPM call

UPM  c1 (InA2_c1, InA1_c1,InA0_c1, InX5_c1, InX2_c1, InX1_c1, InX0_c1, InX4_c1, InX3_c1, InX6_c1, InA4_c1,
        InA3_c1, InD2_c1, FP_clock, bit1, bit2, bit3_c1, ad0_c1, ad1_c1, ad2_c1, ad3_c1, ad4_c1,
        ad5_c1, Out0_c1, Out1_c1, Out4_c1, Out3_c1, Out2_c1, Out5_c1, Out6_c1, result2_c1);  //UPM call

UPM  c2 (InA2_c2, InA1_c2,InA0_c2, InX5_c2, InX2_c2, InX1_c2, InX0_c2, InX4_c2, InX3_c2, InX6_c2, InA4_c2,
        InA3_c2, InD2_c2, FP_clock, bit1, bit2, bit3_c2, ad0_c2, ad1_c2, ad2_c2, ad3_c2, ad4_c2,
        ad5_c2, Out0_c2, Out1_c2, Out4_c2, Out3_c2, Out2_c2, Out5_c2, Out6_c2, result2_c2);  //UPM call

UPM  c3 (InA2_c3, InA1_c3,InA0_c3, InX5_c3, InX2_c3, InX1_c3, InX0_c3, InX4_c3, InX3_c3, InX6_c3, InA4_c3,
        InA3_c3, InD2_c3, FP_clock, bit1, bit2, bit3_c3, ad0_c3, ad1_c3, ad2_c3, ad3_c3, ad4_c3,
        ad5_c3, Out0_c3, Out1_c3, Out4_c3, Out3_c3, Out2_c3, Out5_c3, Out6_c3, result2_c3);  //UPM call

//Appels de fonctions registres de 32-bits indépendants entre eux
Registre32b    regA  (clk0, Samples, clk0, datax1);
Registre32b    regB  (clk, dataI2, clk, datax2);
Registre32b    regC  (clk, dataI3, clk, datax3);
Registre32b    regD  (clk, dataI4, clk, datax4);
Registre32b    regE  (clk, dataI5, clk, datax5);
Registre32b    regF  (clk, dataI6, clk, datax6);
Registre32b    regG  (clk, dataI7, clk, datax7);
Registre32b    regH  (clk, dataI8, clk, datax8);
//Appels de fonctions registres de 32-bits interconnectés entre eux
Registre32b    reg0  (clk0, Samples1, clk0, data1);
Registre32b    reg1  (clk0, data1, clk0, data2);
Registre32b    reg2  (clk0, data2, clk0, data3);
Registre32b    reg3  (clk0, data3, clk0, data4);
Registre32b    reg4  (clk0, data4, clk0, data5);
Registre32b    reg5  (clk0, data5, clk0, data6);
Registre32b    reg6  (clk0, data6, clk0, data7);
Registre32b    reg7  (clk0, data7, clk0, data8);
Registre32b    reg8  (clk0, data8, clk0, data9);
Registre32b    reg9  (clk0, data9, clk0, data10);
Registre32b    reg10 (clk0, data10, clk0, data11);
Registre32b    reg11 (clk0, data11, clk0, data12);
Registre32b    reg12 (clk0, data12, clk0, data13);
Registre32b    reg13 (clk0, data13, clk0, data14);
Registre32b    reg14 (clk0, data14, clk0, data15);
Registre32b    reg15 (clk0, data15, clk0, data16);
Registre32b    reg16 (clk0, data16, clk0, data17);
//Appel d'un multiplexeur de 8x32b pour le contrôle des fonctions à plusieurs sorties
Multiplexeur8x32b  mx0 ( InMux0, InMux1, InMux2, InMux3, InMux4, InMux5, InMux6, InMux7, MuxCtrl, MuxOutput);
//Construction des horloges de contrôle à partir de l'entrée FP_clock
delay          dl0   (FP_clock, 7'b1000001, S0);
clock_divider   cl0   (S0, 5'b01110, 5'b11010, clk_Opcode0);
delay          dl1   (clk_Opcode0, 7'b0000010, S1);
clock_divider   cl1   (S1, 5'b00101, 5'b01000, bit2_Opcode0);
clock_divider0       cl2   (FP_clock,   clk0_Opcode0);
clock_divider1       cl3   (FP_clock,   clk0_Opcode1);
//Appels de trois multiplexeurs de 1x8b qui contrôlent la selection des horloges
```

ANNEXE 15 – Algorithme du processeur matriciel *2x2* UPMs (2)

```
Multiplexeur8x1b  mx1 ( clk_Opcode0, clk_Opcode0,   clk_Opcode0, clk_Opcode0, clk_Opcode0, clk_Opcode0, clk_Opcode0,
clk_Opcode0, Opcode, clk);
Multiplexeur8x1b  mx2 (bit2_Opcode0, bit2_Opcode0, bit2_Opcode0, clk_Opcode0,       1'b0, bit2_Opcode0,
bit2_Opcode0,bit2_Opcode0,  Opcode, bit2);
Multiplexeur8x1b  mx3 (       1'b0, clk0_Opcode0,       1'b0, clk0_Opcode0, clk0_Opcode0, clk0_Opcode0, clk0_Opcode1,
clk0_Opcode1, Opcode, clk0);

//
//Controle temporel du Processeur
//À chaque montée de l'horloge clk
always @ (posedge clk)
begin
////////////////////////////
/////NON-ADAPTIVE
FIR-IIR////
////& CORRELATOR
MODE N=20///
////////////////////////////
  if (Opcode==3'b000)
begin
//NON-ADAPTIVE -IIR
//CORRELATOR N=20
// processor input
  InA2_c0 <= Samples;
//Control bits
bit1        <= 1'b0;
bit3_c0 <= 1'b1;
 bit3_c1 <= 1'b1;
bit3_c2 <= 1'b1;
bit3_c3 <= 1'b1;
    ad0_c0  <= 1'b1;
    ad1_c0  <= 1'b1;
    ad2_c0  <= 1'b1;
    ad3_c0  <= 1'b1;
    ad4_c0  <= 1'b1;
    ad5_c0  <= 1'b0;
    ad0_c1  <= 1'b1;
    ad1_c1  <= 1'b1;
    ad2_c1  <= 1'b1;
    ad3_c1  <= 1'b1;
    ad4_c1  <= 1'b1;
    ad5_c1  <= 1'b0;
    ad0_c2  <= 1'b1;
    ad1_c2  <= 1'b1;
    ad2_c2  <= 1'b1;
    ad3_c2  <= 1'b1;
    ad4_c2  <= 1'b1;
    ad5_c2  <= 1'b0;
    ad0_c3  <= 1'b1;
    ad1_c3  <= 1'b1;
    ad2_c3  <= 1'b1;
    ad3_c3  <= 1'b1;
    ad4_c3  <= 1'b1;
    ad5_c3  <= 1'b0;
//Initialization of
//UPM c0 parameters
InA1_c0  <= zero;
    InA0_c0 <= zero;
    InX5_c0 <= zero;
    InX2_c0 <= un;
    InX1_c0 <= un;
    InX0_c0 <= un;
    InX4_c0 <= zero;
    InX3_c0 <= zero;
    InA4_c0 <= un;
    InA3_c0 <= un;
    InD2_c0 <= zero;
    //Interconnection
    //with neighbourhood
    InX6_c0 <= Out3_c3;
    InX5_c1 <= Out1_c0;
    InD2_c1 <= Out4_c0;
    InX6_c1 <= Out6_c0;
//Initialization
//of UPM c1 parameters
InA2_c1 <= result2_c0;
InA1_c1  <= zero;
    InA0_c1 <= zero;
    InX2_c1 <= un;
    InX1_c1 <= un;
    InX0_c1 <= un;
    InX4_c1 <= zero;
    InX3_c1 <= zero;
    InA4_c1 <= un;
    InA3_c1 <= un;
    //Interconnection
    //with neighbourhood
    InX5_c2 <= Out1_c1;
    InD2_c2 <= Out4_c1;
    InX6_c2 <= Out6_c1;
     //Initialization
    //of UPM c2
///parameters
InA2_c2 <= zero;
InA1_c2  <= zero;
    InA0_c2 <= zero;
    InX2_c2 <= zero;
    InX1_c2 <= zero;
    InX0_c2 <= zero;
    InX4_c2 <= zero;
    InX3_c2 <= zero;
    InA4_c2 <= un;
    InA3_c2 <= un;
       //Interconnection
    //with neighbourhood
    InX5_c3 <= Out1_c2;
    InD2_c3 <= Out4_c2;
    InX6_c3 <= Out6_c2;
    //Initialization of
    //UPM c3 parameters
InA2_c3 <= result2_c1;
InA1_c3  <= zero;
    InA0_c3 <= zero;
    InX2_c3 <= un;
    InX1_c3 <= un;
    InX0_c3 <= un;
    InX4_c3 <= zero;
    InX3_c3 <= zero;
    InA4_c3 <= un;
    InA3_c3 <= un;
     // IIR outputs
    Output0 <= Out1_c3;
    //IIR output
    Output1 <= Out3_c3;
end

////////////////////////////
/////ADAPTIVE FIR-
IIR////////
////& CORRELATOR
MODE N=9///
////////////////////////////
if (Opcode==3'b001)
begin
//ADAPTIVE FIR-IIR
//CORRELATOR N=9
// processor input
InA2_c0 <= Samples;
//Control bits
bit1        <= 1'b0;
bit3_c0 <= 1'b1;
bit3_c1 <= 1'b1;
bit3_c2 <= 1'b1;
bit3_c3 <= 1'b1;
    ad0_c0  <= 1'b1;
    ad1_c0  <= 1'b1;
    ad2_c0  <= 1'b1;
    ad3_c0  <= 1'b1;
    ad4_c0  <= 1'b1;
    ad5_c0  <= 1'b0;
    ad0_c1  <= 1'b1;
    ad1_c1  <= 1'b1;
    ad2_c1  <= 1'b1;
    ad3_c1  <= 1'b1;
    ad4_c1  <= 1'b1;
    ad5_c1  <= 1'b0;
    ad0_c2  <= 1'b1;
    ad1_c2  <= 1'b1;
    ad2_c2  <= 1'b1;
    ad3_c2  <= 1'b1;
    ad4_c2  <= 1'b1;
    ad5_c2  <= 1'b0;
    ad0_c3  <= 1'b1;
    ad1_c3  <= 1'b1;
    ad2_c3  <= 1'b1;
    ad3_c3  <= 1'b1;
    ad4_c3  <= 1'b1;
    ad5_c3  <= 1'b0;
    //Initialization of
    //UPM c0 parameters
InA1_c0  <= zero;
    InA0_c0 <= zero;
    InX5_c0 <= zero;
    InX2_c0 <= data1;
    InX1_c0 <= data2;
    InX0_c0 <= data3;
    InX4_c0 <= zero;
    InX3_c0 <= zero;
    InA4_c0 <= data4;
    InA3_c0 <= data5;
    InD2_c0 <= zero;
    //Interconnection
    //with neighbourhood
    InX6_c0 <= Out3_c3;
    InX5_c1 <= Out1_c0;
    InD2_c1 <= Out4_c0;
    InX6_c1 <= Out6_c0;
    //Initialization
 //of UPM c1 parameters
InA2_c1 <= result2_c0;
InA1_c1  <= zero;
    InA0_c1 <= zero;
    InX2_c1 <= data6;
    InX1_c1 <= data7;
    InX0_c1 <= data8;
    InX4_c1 <= zero;
    InX3_c1 <= zero;
    InA4_c1 <= data9;
    InA3_c1 <= data10;
    //Interconnection
```

ANNEXE 16 – Algorithme du processeur matriciel *2x2* UPMs (3)

```
  //with neighbourhood
    InX5_c2 <= Out1_c1;
    InD2_c2 <= Out4_c1;
    InX6_c2 <= Out6_c1;
     //Initialization
    //of UPM c2
InA2_c2 <= zero;
InA1_c2  <= zero;
    InA0_c2 <= zero;
    InX2_c2 <= zero;
    InX1_c2 <= zero;
    InX0_c2 <= zero;
    InX4_c2 <= zero;
    InX3_c2 <= zero;
    InA4_c2 <= data11;
    InA3_c2 <= data12;
//Interconnection
//with neighbourhood
    InX5_c3 <= Out1_c2;
    InD2_c3 <= Out4_c2;
    InX6_c3 <= Out6_c2;
    //Initialization of
//UPM c3 parameters
InA2_c3 <= result2_c1;
InA1_c3  <= zero;
    InA0_c3 <= zero;
    InX2_c3 <= data13;
    InX1_c3 <= data14;
    InX0_c3 <= data15;
    InX4_c3 <= zero;
    InX3_c3 <= zero;
    InA4_c3 <= data16;
    InA3_c3 <= data17;
     //FIR outputs
    Output0 <= Out1_c3;
    //IIR output delayed
    Output1 <= Out3_c3;
end

//////////////////////////////
/////ALL-POLE
LATTICE MODE///
//////////////////////////////
else  if
(Opcode==3'b010)
begin
 //ALL-POLE LATTICE
//Processor input
InX2_c0 <= Samples;
//Control bits
bit1 <= 1'b1;
bit3_c0 <= 1'b1;
bit3_c1 <= 1'b1;
bit3_c2 <= 1'b1;
bit3_c3 <= 1'b1;
//Add/Sub bits
    ad0_c0  <= 1'b1;
    ad1_c0  <= 1'b0;
    ad2_c0  <= 1'b1;
    ad3_c0  <= 1'b1;
    ad4_c0  <= 1'b1;
    ad5_c0  <= 1'b0;
    ad0_c1  <= 1'b1;
    ad1_c1  <= 1'b0;
    ad2_c1  <= 1'b1;
    ad3_c1  <= 1'b1;
    ad4_c1  <= 1'b1;
    ad5_c1  <= 1'b0;
    ad0_c2  <= 1'b1;
    ad1_c2  <= 1'b0;
    ad2_c2  <= 1'b1;
    ad3_c2  <= 1'b1;
    ad4_c2  <= 1'b1;
    ad5_c2  <= 1'b0;
    ad0_c3  <= 1'b1;
    ad1_c3  <= 1'b0;
    ad2_c3  <= 1'b1;
    ad3_c3  <= 1'b1;
    ad4_c3  <= 1'b1;
    ad5_c3  <= 1'b0;
//Initialization of
//UPM c0 parameters
InA2_c0 <= un;
InA1_c0  <= un;
    InA0_c0 <= zero;
    InX5_c0 <= zero;
    InX0_c0 <= zero;
    InA4_c0 <= un;
    InA3_c0 <= un;
    InD2_c0 <= zero;
    //Interconnection
    //with neighbourhood
    InX1_c0 <= Out5_c0;
    InX4_c0 <= Out5_c0;
    InX3_c0 <= Out0_c0;
    InX6_c0 <= Out4_c1;
    //Next UPM input
    InX2_c1 <= Out0_c0;
    //Initialization of
    //UPM c1 parameters
InA2_c1 <= un;
InA1_c1  <= un;
    InA0_c1 <= zero;
    InX5_c1 <= zero;
    InX0_c1 <= zero;
    InA4_c1 <= un;
    InA3_c1 <= un;
    InD2_c1 <= zero;
     //Interconnection
    //with neighbourhood
    InX1_c1 <= Out5_c1;
    InX4_c1 <= Out5_c1;
    InX3_c1 <= Out0_c1;
    InX6_c1 <= Out4_c2;
    //Next UPM input
    InX2_c2 <= Out0_c1;
//Initialization of
//UPM c2 parameters
InA2_c2 <= un;
InA1_c2  <= un;
    InA0_c2 <= zero;
    InX5_c2 <= zero;
    InX0_c2 <= zero;
    InA4_c2 <= un;
    InA3_c2 <= un;
    InD2_c2 <= zero;
     //Interconnection
    //with neighbourhood
    InX1_c2 <= Out5_c2;
    InX4_c2 <= Out5_c2;
    InX3_c2 <= Out0_c2;
    InX6_c2 <= Out4_c3;
    //Next UPM input
    InX2_c3 <= Out0_c2;
    //Initialization of
    //UPM c2 parameters
InA2_c3 <= un;
InA1_c3  <= un;
    InA0_c3 <= zero;
    InX5_c3 <= zero;
    InX0_c3 <= zero;
    InA4_c3 <= un;
    InA3_c3 <= un;
    InD2_c3 <= zero;
//Interconnection
//with neighbourhood
    InX1_c3 <= Out5_c3;
    InX4_c3 <= Out5_c3;
    InX3_c3 <= Out0_c3;
    InX6_c3 <= Out0_c3;
    //ALL-POLE
LATTICE
    //processor output
    Output0 <= Out0_c3;
    Output1 <= Out4_c3;
end

//////////////////////////////
/////ALL-ZERO
LATTICE MODE///
//////////////////////////////
 else  if
(Opcode==3'b011)
begin
     //Processor inputs
    InX2_c0 <= Samples;
    InX1_c0 <= datax1;
    InX4_c0 <= datax1;
    InX3_c0 <= Samples;
//Control bits
bit1 <= 1'b1;
bit3_c0 <= 1'b1;
bit3_c1 <= 1'b1;
bit3_c2 <= 1'b1;
bit3_c3 <= 1'b1;
//Add/Sub bits
    ad0_c0  <= 1'b1;
    ad1_c0  <= 1'b1;
    ad2_c0  <= 1'b1;
    ad3_c0  <= 1'b1;
    ad4_c0  <= 1'b1;
    ad5_c0  <= 1'b1;
    ad0_c1  <= 1'b1;
    ad1_c1  <= 1'b1;
    ad2_c1  <= 1'b1;
    ad3_c1  <= 1'b1;
    ad4_c1  <= 1'b1;
    ad5_c1  <= 1'b1;
    ad0_c2  <= 1'b1;
    ad1_c2  <= 1'b1;
    ad2_c2  <= 1'b1;
    ad3_c2  <= 1'b1;
    ad4_c2  <= 1'b1;
    ad5_c2  <= 1'b1;
    ad0_c3  <= 1'b1;
    ad1_c3  <= 1'b1;
    ad2_c3  <= 1'b1;
    ad3_c3  <= 1'b1;
    ad4_c3  <= 1'b1;
    ad5_c3  <= 1'b1;
    //Initialization of
    //UPM c0 parameters
InA2_c0 <= un;
InA1_c0  <= un;
    InA0_c0 <= zero;
    InX5_c0 <= zero;
    InX0_c0 <= zero;
    InA4_c0 <= un;
    InA3_c0 <= un;
    InD2_c0 <= zero;
    //Interconnection
    //with neighbourhood
    InX6_c0 <= Out4_c0;
    //Next UPM inputs
    InX2_c1 <= Out0_c0;
    InX3_c1 <= Out0_c0;
    InX1_c1 <= Out5_c0;
    InX4_c1 <= Out5_c0;
    //UPM c1 parameters
InA2_c1 <= un;
InA1_c1  <= un;
    InA0_c1 <= zero;
    InX5_c1 <= zero;
    InX0_c1 <= zero;
    InA4_c1 <= un;
    InA3_c1 <= un;
    InD2_c1 <= zero;
     //Interconnection
    //with neighbourhood
    InX6_c1 <= Out4_c1;
     //Next UPM inputs
    InX2_c2 <= Out0_c1;
    InX3_c2 <= Out0_c1;
    InX1_c2 <= Out5_c1;
    InX4_c2 <= Out5_c1;
//Initialization of
//UPM c2 parameters
InA2_c2 <= un;
InA1_c2  <= un;
```

ANNEXE 17 – Algorithme du processeur matriciel *2*x*2* UPMs (4)

```
InA0_c2 <= zero;
   InX5_c2 <= zero;
   InX0_c2 <= zero;
   InA4_c2 <= un;
   InA3_c2 <= un;
   InD2_c2 <= zero;
    //Interconnection
   //with neighbourhood
   InX6_c2 <= Out4_c2;
   //Next UPM inputs
   InX2_c3 <= Out0_c2;
   InX3_c3 <= Out0_c2;
   InX1_c3 <= Out5_c2;
   InX4_c3 <= Out5_c2;
   //UPM c3 parameters
InA2_c3 <= un;
InA1_c3  <= un;
   InA0_c3 <= zero;
   InX5_c3 <= zero;
   InX0_c3 <= zero;
   InA4_c3 <= un;
   InA3_c3 <= un;
   InD2_c3 <= zero;
    //Interconnection
   //with neighbourhood
   InX6_c3 <= Out4_c3;
    //ALL-ZERO
   Output0 <= Out0_c3;
   Output1 <= Out4_c3;
   end

//////////////////////////
/////Radix-4 Butterfly///
//////////////////////////
else if
(Opcode==3'b100)
begin
//Butterfly r=4 mode
//Control bits//
bit1 <= 1'b1;
bit3_c0 <= 1'b0;
bit3_c1 <= 1'b0;
bit3_c2 <= 1'b0;
bit3_c3 <= 1'b0;
//C0 add_sub bits
ad0_c0  <= 1'b1;
   ad1_c0  <= 1'b1;
   ad2_c0  <= 1'b0;
   ad3_c0  <= 1'b1;
   ad4_c0  <= 1'b1;
   ad5_c0  <= 1'b1;
   ////C1 add_sub bit
   ad0_c1  <= 1'b1;
   ad1_c1  <= 1'b1;
   ad2_c1  <= 1'b0;
   ad3_c1  <= 1'b1;
   ad4_c1  <= 1'b1;
   ad5_c1  <= 1'b1;
   //C2 add_sub bit
   ad0_c2  <= 1'b0;
   ad1_c2  <= 1'b1;
   ad2_c2  <= 1'b0;
   ad3_c2  <= 1'b0;
   ad4_c2  <= 1'b1;
   ad5_c2  <= 1'b1;
   //C3 add_sub bit
   ad0_c3  <= 1'b0;
   ad1_c3  <= 1'b1;
   ad2_c3  <= 1'b0;
   ad3_c3  <= 1'b0;
   ad4_c3  <= 1'b1;
   ad5_c3  <= 1'b1;
//Configuration of
// 2 UPMs c0 and c1
InA2_c0 <= un;
InA1_c0  <= un;
InA0_c0 <= two;
InX5_c0 <= data8;
//R(f0)
InX2_c0 <= data4;
//R(f2)
InX1_c0 <= Out4_c0;
//R(s1)
   InX0_c0 <= Out4_c0;
//R(s1)
   InX4_c0 <= data6;
//R(f1)
   InX3_c0 <= data2;
//R(f3)
   InA4_c0 <= un;
   InA3_c0 <= un;
   InD2_c0 <= zero;
   InX6_c0 <= zero;
   InA2_c1 <= un;
InA1_c1  <= un;
   InA0_c1 <= two;
   InX5_c1 <= data7;
//I(f0)
   InX2_c1 <= data3;
//I(f2)
   InX1_c1 <= Out4_c1;
//I(s1)
   InX0_c1 <= Out4_c1;
//I(s1)
   InX4_c1 <= data5;
//I(f1)
   InX3_c1 <= data1;
//I(f3)
   InA4_c1 <= un;
   InA3_c1 <= un;
   InD2_c1 <= zero;
   InX6_c1 <= zero;
//Configuration of
   // 2 UPMs c2 and c3
   InA2_c2 <= un;
InA1_c2  <= moinsun;
InA0_c2 <= moinstwo;
InX5_c2 <= data8;
//R(f0)
   InX2_c2 <= data4;
//R(f2)
   InX1_c2 <= Out4_c3;
//I(s3)
   InX0_c2 <= Out4_c3;
//I(s3)
   InX4_c2 <= data6;
//R(f1)
   InX3_c2 <= data2;
//R(f3)
   InA4_c2 <= un;
   InA3_c2 <= un;
   InD2_c2 <= zero;
   InX6_c2 <= zero;
   //
   InA2_c3 <= un;
InA1_c3  <= un;
   InA0_c3 <= two;
   InX5_c3 <= data7;
//I(f0)
   InX2_c3 <= data3;
//I(f2)
   InX1_c3 <=
Out4_c2;//R(s3)
   InX0_c3 <=
Out4_c2;//R(s3)
   InX4_c3 <= data5;
//I(f1)
   InX3_c3 <= data1;
//I(f3)
   InA4_c3 <= un;
   InA3_c3 <= un;
   InD2_c3 <= zero;
   InX6_c3 <= zero;
   //r=4 Butterfly
outputs
   InMux0 <= Out0_c0;
//R(F0)
   InMux1 <= Out0_c1;
//I(F0)
   InMux2 <= Out1_c2;
//R(F1)
   InMux3 <= Out1_c3;
//I(F1)
   InMux4 <= Out1_c0;
//R(F2)
   InMux5 <= Out1_c1;
//I(F2)
   InMux6 <= Out0_c2;
//R(F3)
   InMux7 <= Out0_c3;
//I(F3)
Output0 <= MuxOutput;
Output1 <= zero;
end

 //////////////////////////
/////DIVISION////////////
//////////////////////////
else  if
(Opcode==3'b101)
begin
//Control bits
bit1 <= 1'b1;
   bit3_c0 <= 1'b1;
   bit3_c1 <= 1'b1;
bit3_c2 <= 1'b1;
bit3_c3 <= 1'b1;
//Add/Sub bits
   ad0_c0  <= 1'b1;
   ad1_c0  <= 1'b0;
   ad2_c0  <= 1'b1;
   ad3_c0  <= 1'b1;
   ad4_c0  <= 1'b1;
   ad5_c0  <= 1'b0;
   ad0_c1  <= 1'b1;
   ad1_c1  <= 1'b0;
   ad2_c1  <= 1'b1;
   ad3_c1  <= 1'b1;
   ad4_c1  <= 1'b1;
   ad5_c1  <= 1'b0;
   ad0_c2  <= 1'b1;
   ad1_c2  <= 1'b0;
   ad2_c2  <= 1'b1;
   ad3_c2  <= 1'b1;
   ad4_c2  <= 1'b1;
   ad5_c2  <= 1'b0;
   ad0_c3  <= 1'b1;
   ad1_c3  <= 1'b0;
   ad2_c3  <= 1'b1;
   ad3_c3  <= 1'b1;
   ad4_c3  <= 1'b1;
   ad5_c3  <= 1'b0;
   //Initialization of
   //UPM c0 parameters
InA2_c0 <= Samples1;
InA1_c0  <= Samples;
 InA0_c0 <= Samples1;
   InX5_c0 <= zero;
   InX2_c0 <= two;
   InX1_c0 <= Out2_c0;
 InX0_c0 <= Samples1;
   InA4_c0 <= zero;
   InA3_c0 <= zero;
   InD2_c0 <= zero;
   InX4_c0 <= zero;
   InX3_c0 <= zero;
   InX6_c0 <= zero;
   //Initialization of
   //UPM c1 parameters
InA2_c1 <= Out0_c0;
InA1_c1  <= Samples;
   InA0_c1 <= Out0_c0;
   InX5_c1 <= zero;
   InX2_c1 <= two;
   InX1_c1 <= Out2_c1;
   InX0_c1 <= Out0_c0;
   InA4_c1 <= zero;
```

ANNEXE 18 – Algorithme du processeur matriciel *2x2* UPMs (5)

```
InA3_c1 <= zero;
   InD2_c1 <= zero;
   InX4_c1 <= zero;
   InX3_c1 <= zero;
   InX6_c1 <= zero;
   //Initialization of
   //UPM c2 parameters
InA2_c2 <= Out0_c1;
InA1_c2  <= Samples;
   InA0_c2 <= Out0_c1;
   InX5_c2 <= zero;
   InX2_c2 <= two;
   InX1_c2 <= Out2_c2;
   InX0_c2 <= Out0_c1;
   InA4_c2 <= zero;
   InA3_c2 <= zero;
   InD2_c2 <= zero;
   InX4_c2 <= zero;
   InX3_c2 <= zero;
   InX6_c2 <= zero;
   //Initialization of
   //UPM c2 parameters
InA2_c3 <= Out0_c2;
InA1_c3  <= Samples;
   InA0_c3 <= Out0_c2;
   InX5_c3 <= zero;
   InX2_c3 <= two;
   InX1_c3 <= Out2_c3;
   InX0_c3 <= Out0_c2;
   InA4_c3 <= zero;
   InA3_c3 <= zero;
   InD2_c3 <= zero;
   InX4_c3 <= zero;
   InX3_c3 <= zero;
   InX6_c3 <= zero;
   //DIVISION
   //processor output
   Output0 <= Out0_c3;
   dataI1 <= Out0_c3;
   Output1 <= zero;
end
```

ANNEXE 19 – Algorithme du processeur matriciel *6*x4 UPMs (1)

```
////////////////////////////////////UPM_Arrays/////////////////////////////////////////////////
//Top level module instantiation
module UPM_Arrays ( FP_clock, Opcode, MuxCtrl, Samples, Samples1, Output0, Output1, clk, clk0, bit2);
//I/O instantiations
input             FP_clock; //Clock and control inputs
input   [31:0]  Samples, Samples1;       //Array processor input
input   [2:0]    Opcode;                                          //Opcode input
input   [2:0]    MuxCtrl;            //Multiplexer control
output  [31:0]   Output0, Output1;    //Array processor outputs
reg     [31:0]  Output0, Output1;         //Array processor outputs
output           clk, clk0, bit2;
wire             clk, clk0, bit2;
//Internal registers
reg    [31:0]  InA2_c0, InA2_c1, InA2_c2, InA2_c3, InA2_c4, InA2_c5, InA2_c6, InA2_c7, InA2_c8, InA2_c9;
reg    [31:0]  InA2_c10, InA2_c11, InA2_c12, InA2_c13, InA2_c14, InA2_c15, InA2_c16, InA2_c17, InA2_c18;
reg    [31:0]  InA2_c19, InA2_c20, InA2_c21, InA2_c22, InA2_c23;
reg    [31:0]  InA1_c0, InA1_c1, InA1_c2, InA1_c3, InA1_c4, InA1_c5, InA1_c6, InA1_c7, InA1_c8, InA1_c9;
reg    [31:0]  InA1_c10, InA1_c11, InA1_c12, InA1_c13, InA1_c14, InA1_c15, InA1_c16, InA1_c17, InA1_c18;
reg    [31:0]  InA1_c19, InA1_c20, InA1_c21, InA1_c22, InA1_c23;
reg    [31:0]  InA0_c0, InA0_c1, InA0_c2, InA0_c3, InA0_c4, InA0_c5, InA0_c6, InA0_c7, InA0_c8, InA0_c9;
reg    [31:0]  InA0_c10, InA0_c11, InA0_c12, InA0_c13, InA0_c14, InA0_c15, InA0_c16, InA0_c17, InA0_c18;
reg    [31:0]  InA0_c19, InA0_c20, InA0_c21, InA0_c22, InA0_c23;
reg    [31:0]  InX5_c0, InX5_c1, InX5_c2, InX5_c3, InX5_c4, InX5_c5, InX5_c6, InX5_c7, InX5_c8, InX5_c9;
reg    [31:0]  InX5_c10, InX5_c11, InX5_c12, InX5_c13, InX5_c14, InX5_c15, InX5_c16, InX5_c17, InX5_c18;
reg    [31:0]  InX5_c19, InX5_c20, InX5_c21, InX5_c22, InX5_c23;
reg    [31:0]  InX2_c0, InX2_c1, InX2_c2, InX2_c3, InX2_c4, InX2_c5, InX2_c6, InX2_c7, InX2_c8, InX2_c9;
reg    [31:0]  InX2_c10, InX2_c11, InX2_c12, InX2_c13, InX2_c14, InX2_c15, InX2_c16, InX2_c17, InX2_c18;
reg    [31:0]  InX2_c19, InX2_c20, InX2_c21, InX2_c22, InX2_c23;
reg    [31:0]  InX1_c0, InX1_c1, InX1_c2, InX1_c3, InX1_c4, InX1_c5, InX1_c6, InX1_c7, InX1_c8, InX1_c9;
reg    [31:0]  InX1_c10, InX1_c11, InX1_c12, InX1_c13, InX1_c14, InX1_c15, InX1_c16, InX1_c17, InX1_c18;
reg    [31:0]  InX1_c19, InX1_c20, InX1_c21, InX1_c22, InX1_c23;
reg    [31:0]  InX0_c0, InX0_c1, InX0_c2, InX0_c3, InX0_c4, InX0_c5, InX0_c6, InX0_c7, InX0_c8, InX0_c9;
reg    [31:0]  InX0_c10, InX0_c11, InX0_c12, InX0_c13, InX0_c14, InX0_c15, InX0_c16, InX0_c17, InX0_c18;
reg    [31:0]  InX0_c19, InX0_c20, InX0_c21, InX0_c22, InX0_c23;
reg    [31:0]  InX4_c0, InX4_c1, InX4_c2, InX4_c3, InX4_c4, InX4_c5, InX4_c6, InX4_c7, InX4_c8, InX4_c9;
reg    [31:0]  InX4_c10, InX4_c11, InX4_c12, InX4_c13, InX4_c14, InX4_c15, InX4_c16, InX4_c17, InX4_c18;
reg    [31:0]  InX4_c19, InX4_c20, InX4_c21, InX4_c22, InX4_c23;
reg    [31:0]  InX3_c0, InX3_c1, InX3_c2, InX3_c3, InX3_c4, InX3_c5, InX3_c6, InX3_c7, InX3_c8, InX3_c9;
reg    [31:0]  InX3_c10, InX3_c11, InX3_c12, InX3_c13, InX3_c14, InX3_c15, InX3_c16, InX3_c17, InX3_c18;
reg    [31:0]  InX3_c19, InX3_c20, InX3_c21, InX3_c22, InX3_c23;
reg    [31:0]  InA4_c0, InA4_c1, InA4_c2, InA4_c3, InA4_c4, InA4_c5, InA4_c6, InA4_c7, InA4_c8, InA4_c9;
reg    [31:0]  InA4_c10, InA4_c11, InA4_c12, InA4_c13, InA4_c14, InA4_c15, InA4_c16, InA4_c17, InA4_c18;
reg    [31:0]  InA4_c19, InA4_c20, InA4_c21, InA4_c22, InA4_c23, InX6_c10, InD2_c10;
reg    [31:0]  InA3_c0, InA3_c1, InA3_c2, InA3_c3, InA3_c4, InA3_c5, InA3_c6, InA3_c7, InA3_c8, InA3_c9;
reg    [31:0]  InA3_c10, InA3_c11, InA3_c12, InA3_c13, InA3_c14, InA3_c15, InA3_c16, InA3_c17, InA3_c18;
reg    [31:0]  InA3_c19, InA3_c20, InA3_c21, InA3_c22, InA3_c23, InX6_c11, InD2_c11;
reg    [31:0]  InD2_c0, InD2_c1, InD2_c2, InD2_c3, InD2_c4, InD2_c5, InD2_c6, InD2_c7, InD2_c8, InD2_c9;
reg    [31:0]  InX6_c0, InX6_c1, InX6_c2, InX6_c3, InX6_c4, InX6_c5, InX6_c6, InX6_c7, InX6_c8, InX6_c9;
reg          bit3_c0, bit3_c1, bit3_c2, bit3_c3, bit3_c4, bit3_c5, bit3_c6, bit3_c7, bit3_c8, bit3_c9;
reg          bit1, ad0_c0, ad0_c1, ad0_c2, ad0_c3, ad0_c4, ad0_c5, ad0_c6, ad0_c7, ad0_c8, ad0_c9;
reg          ad0_c10, ad0_c11, ad0_c12, ad0_c13, ad0_c14, ad0_c15, ad0_c16, ad0_c17, ad0_c18, ad0_c19;
reg          ad0_c20, ad0_c21, ad0_c22, ad0_c23;
reg          ad1_c0, ad1_c1, ad1_c2, ad1_c3, ad1_c4, ad1_c5, ad1_c6, ad1_c7, ad1_c8, ad1_c9, ad1_c10;
reg          ad1_c11, ad1_c12, ad1_c13, ad1_c14, ad1_c15, ad1_c16, ad1_c17, ad1_c18, ad1_c19, ad1_c20;
reg          ad1_c21, ad1_c22, ad1_c23;
reg          ad2_c0, ad2_c1, ad2_c2, ad2_c3, ad2_c4, ad2_c5, ad2_c6, ad2_c7, ad2_c8, ad2_c9, ad2_c10;
reg          ad2_c11, ad2_c12, ad2_c13, ad2_c14, ad2_c15, ad2_c16, ad2_c17, ad2_c18, ad2_c19, ad2_c20;
reg          ad2_c21, ad2_c22, ad2_c23;
```

ANNEXE 20 – Algorithme du processeur matriciel *6*x4 UPMs (2)

```
reg         ad3_c0, ad3_c1, ad3_c2, ad3_c3, ad3_c4, ad3_c5, ad3_c6, ad3_c7, ad3_c8, ad3_c9, ad3_c10;
reg         ad3_c11, ad3_c12, ad3_c13, ad3_c14, ad3_c15, ad3_c16, ad3_c17, ad3_c18, ad3_c19, ad3_c20;
reg         ad3_c21, ad3_c22, ad3_c23;
reg         ad4_c0, ad4_c1, ad4_c2, ad4_c3, ad4_c4, ad4_c5, ad4_c6, ad4_c7, ad4_c8, ad4_c9;
reg         ad5_c0, ad5_c1, ad5_c2, ad5_c3, ad5_c4, ad5_c5, ad5_c6, ad5_c7, ad5_c8, ad5_c9;
reg   [31:0]  OutMux0, OutMux10, OutMux11, OutMux12, OutMux13, OutMux14, OutMux15, OutMux16;
reg   [31:0]  OutMux17, OutMux18, OutMux19, OutMux1, OutMux20, OutMux21, OutMux22, OutMux23, OutMux24;
reg   [31:0]  OutMux25, OutMux26, OutMux27, OutMux28, OutMux29, OutMux2, OutMux30, OutMux31, OutMux3;
reg   [31:0]  OutMux4, OutMux5, OutMux6, OutMux7, OutMux8, OutMux9, LatchA1, LatchA2, LatchA3, LatchA0;
reg   [31:0]  LatchA4, LatchA5, LatchA6, LatchA7, LatchA8, LatchA9, LatchA10, LatchA11, LatchA12,LatchA13;
reg   [31:0]  LatchA14, LatchA15, LatchA16, LatchA17, LatchA18, LatchA19, LatchA20, LatchA21, LatchA22;
reg   [31:0]  LatchA23, LatchA24, LatchA25, LatchA26, LatchA27, LatchA28, LatchA29, LatchA30, LatchA31;

//Internal connections
wire    [31:0]  Out0_c0, Out0_c1, Out0_c2, Out0_c3, Out0_c4, Out0_c5, Out0_c6, Out0_c7, Out0_c8, Out0_c9;
wire    [31:0]  Out0_c10, Out0_c11, Out0_c12, Out0_c13, Out0_c14, Out0_c15, Out0_c16, Out0_c17, Out0_c18;
wire    [31:0]  Out0_c19, Out0_c20, Out0_c21, Out0_c22, Out0_c23;
wire    [31:0]  Out1_c0, Out1_c1, Out1_c2, Out1_c3, Out1_c4, Out1_c5, Out1_c6, Out1_c7, Out1_c8, Out1_c9;
wire    [31:0]  Out1_c10, Out1_c11, Out1_c12, Out1_c13, Out1_c14, Out1_c15, Out1_c16, Out1_c17, Out1_c18;
wire    [31:0]  Out1_c19, Out1_c20, Out1_c21, Out1_c22, Out1_c23;
wire    [31:0]  Out2_c0, Out2_c1, Out2_c2, Out2_c3, Out2_c4, Out2_c5, Out2_c6, Out2_c7, Out2_c8, Out2_c9;
wire    [31:0]  Out2_c10, Out2_c11, Out2_c12, Out2_c13, Out2_c14, Out2_c15, Out2_c16, Out2_c17, Out2_c18;
wire    [31:0]  Out2_c19, Out2_c20, Out2_c21, Out2_c22, Out2_c23;
wire    [31:0]  Out3_c0, Out3_c1, Out3_c2, Out3_c3, Out3_c4, Out3_c5, Out3_c6, Out3_c7, Out3_c8, Out3_c9;
wire    [31:0]  Out4_c0, Out4_c1, Out4_c2, Out4_c3, Out4_c4, Out4_c5, Out4_c6, Out4_c7, Out4_c8, Out4_c9;
wire    [31:0]  Out4_c10, Out4_c11, Out4_c12, Out4_c13, Out4_c14, Out4_c15, Out4_c16, Out4_c17, Out4_c18;
wire    [31:0]  Out4_c19, Out4_c20, Out4_c21, Out4_c22, Out4_c23;
wire    [31:0]  Out5_c0, Out5_c1, Out5_c2, Out5_c3, Out5_c4, Out5_c5, Out5_c6, Out5_c7, Out5_c8, Out5_c9;
wire    [31:0]  Out6_c0, Out6_c1, Out6_c2, Out6_c3, Out6_c4, Out6_c5, Out6_c6, Out6_c7, Out6_c8, Out6_c9;
wire    [31:0]  result2_c0, result2_c1, result2_c2, result2_c3, result2_c4, result2_c5, result2_c6;
wire    [31:0]  result2_c7, result2_c8, result2_c9, result2_c10, result2_c11, result2_c12, result2_c13;
wire    [31:0]  result2_c14, result2_c15, result2_c16, result2_c17, result2_c18, result2_c19, result2_c20;
wire    [31:0]  result2_c21, result2_c22, result2_c23;
wire    [31:0]  data0, data1, data2, data3, data4, data5, data6, data7, data8;
wire    [31:0]  data9, data10, data11, data12, data13, data14, data15;
wire      [31:0]    MuxOutput, datax1, data16, data17, data18;
wire    [31:0]  data19, data20, data21, data22, data23, data24, data25;
wire    [31:0]  data26, data27, data28, data1x;
wire    [31:0]  data29, data30, data31, data32, data33;
wire    [31:0]      LatchB1,LatchB2,LatchB3, LatchB4, LatchB5, LatchB6, LatchB7, LatchB8, LatchB9, LatchB10;
wire      [31:0]    LatchB11, LatchB12, LatchB13, LatchB0;
wire    [31:0]  LatchB14, LatchB15, LatchB16, LatchB17, LatchB18, LatchB19, LatchB20, LatchB21, LatchB22;
wire    [31:0]  LatchB23, LatchB24, LatchB25, LatchB26, LatchB27, LatchB28, LatchB29, LatchB30, LatchB31;
wire            S0, S1, S2, clk_Opcode0, bit2_Opcode0, clk0_Opcode0, clk0_Opcode2;
//Parameter instantiations
parameter       un          =       32'b00111111100000000000000000000000;  //=1
parameter       moinsun  =          32'b10111111100000000000000000000000;  //=-1
parameter       zero     =          32'b00000000000000000000000000000000;  //=0
parameter       two      =          32'b01000000000000000000000000000000;  //=2
parameter       moinstwo=         32'b11000000000000000000000000000000;  //=-2
parameter  RW1   =    32'b00111111011011001000010010110110;  //=0.9239 - 0.3827i
parameter  IW1   =    32'b10111110110000111111000101000001;
parameter  RW2   =    32'b00111111001101010000010010000001;  //0.7071 - 0.7071i
parameter  IW2   =    32'b10111111001101010000010010000001;
parameter  RW3   =    32'b00111110110000111111000101000001;  //0.3827 - 0.9239i
parameter  IW3   =    32'b10111111011011001000010010110110;
parameter  RW6   =    32'b10111111001101010000010010000001;  //-0.7071 - 0.7071i
parameter  IW6   =    32'b10111111001101010000010010000001;
```

ANNEXE 21 – Algorithme du processeur matriciel *6*x4 UPMs (3)

```
parameter   RW9    =      32'b10111111011011001000010010110110;   //-0.9239 + 0.3827i
parameter   IW9    =      32'b00111110110000111111000101000001;
parameter   zero_p_cinq=      32'b00111111000000000000000000000000;  //0.5
//////////////////////////////////////////////////////////////////
////////////////////FUNCTION CALLs/////////////////////////////////

///Interconnected registers
Registre32b    l0 (clk0, Samples1, clk0, data0);
Registre32b    l1 (clk0, data0, clk0, data1);
Registre32b    l2 (clk0, data1, clk0, data2);
Registre32b    l3 (clk0, data2, clk0, data3);          //Register function
Registre32b    l4 (clk0, data3, clk0, data4);          // Register function
Registre32b    l5 (clk0, data4, clk0, data5);          // Register function
Registre32b    l6 (clk0, data5, clk0, data6);          // Register function
Registre32b    l7 (clk0, data6, clk0, data7);          // Register function
Registre32b    l8 (clk0, data7, clk0, data8);          // Register function
Registre32b    l9 (clk0, data8, clk0, data9);          // Register function
Registre32b    l10 (clk0, data9, clk0, data10); // Register function
Registre32b    l11 (clk0, data10, clk0, data11); // Register function
Registre32b    l12 (clk0, data11, clk0, data12); // Register function
Registre32b    l13 (clk0, data12, clk0, data13); // Register function
Registre32b    l14 (clk0, data13, clk0, data14); // Register function
Registre32b    l15 (clk0, data14, clk0, data15); // Register function
Registre32b    l16 (clk0, data15, clk0, data16); // Register function
Registre32b    l17 (clk0, data16, clk0, data17); // Register function
Registre32b    l18 (clk0, data17, clk0, data18); // Register function
Registre32b    l19 (clk0, data18, clk0, data19); // Register function
Registre32b    l20 (clk0, data19, clk0, data20); // Register function
Registre32b    l21 (clk0, data20, clk0, data21); // Register function
Registre32b    l22 (clk0, data21, clk0, data22); // Register function
Registre32b    l23 (clk0, data22, clk0, data23); // Register function
Registre32b    l24 (clk0, data23, clk0, data24); // Register function
Registre32b    l25 (clk0, data24, clk0, data25); // Register function
Registre32b    l26 (clk0, data25, clk0, data26); // Register function
Registre32b    l27 (clk0, data26, clk0, data27); // Register function
Registre32b    l28 (clk0, data27, clk0, data28); // Register function
Registre32b    l29 (clk0, data28, clk0, data29); // Register function
Registre32b    l30 (clk0, data29, clk0, data30); // Register function
Registre32b    l31 (clk0, data30, clk0, data31); // Register function

//Independent Registers

Registre32b    lA (clk0, Samples, clk0, data1x);

Registre32b    lA0 (clk0, LatchA0, clk0, LatchB0);
Registre32b    lA1 (clk0, LatchA1, clk0, LatchB1);
Registre32b    lA2 (clk0, LatchA2, clk0, LatchB2);
Registre32b    lA3 (clk0, LatchA3, clk0, LatchB3);          // Register function
Registre32b    lA4 (clk0,LatchA4, clk0, LatchB4);           // Register function
Registre32b    lA5 (clk0, LatchA5, clk0, LatchB5);          // Register function
Registre32b    lA6 (clk0, LatchA6, clk0, LatchB6);          // Register function
Registre32b    lA7 (clk0, LatchA7, clk0, LatchB7);          // Register function
Registre32b    lA8 (clk0, LatchA8, clk0, LatchB8);          // Register function
Registre32b    lA9 (clk0, LatchA9, clk0, LatchB9);          // Register function
Registre32b    lA10 (clk0, LatchA10, clk0, LatchB10);
Registre32b    lA11 (clk0, LatchA11, clk0, LatchB11);
Registre32b    lA12 (clk0, LatchA12, clk0, LatchB12);
Registre32b    lA13 (clk0, LatchA13, clk0, LatchB13);
```

ANNEXE 22 – Algorithme du processeur matriciel *6*x4 UPMs (4)

```
Registre32b     lA14 (clk0, LatchA14, clk0, LatchB14);
Registre32b     lA15 (clk0, LatchA15, clk0, LatchB15);
Registre32b     lA16 (clk0, LatchA16, clk0, LatchB16);
Registre32b     lA17 (clk0, LatchA17, clk0, LatchB17);
Registre32b     lA18 (clk0, LatchA18, clk0, LatchB18);
Registre32b     lA19 (clk0, LatchA19, clk0, LatchB19);
Registre32b     lA20 (clk0, LatchA20, clk0, LatchB20);
Registre32b     lA21 (clk0, LatchA21, clk0, LatchB21);
Registre32b     lA22 (clk0, LatchA22, clk0, LatchB22);
Registre32b     lA23 (clk0, LatchA23, clk0, LatchB23);
Registre32b     lA24 (clk0, LatchA24, clk0, LatchB24);
Registre32b     lA25 (clk0, LatchA25, clk0, LatchB25);
Registre32b     lA26 (clk0, LatchA26, clk0, LatchB26);
Registre32b     lA27 (clk0, LatchA27, clk0, LatchB27);
Registre32b     lA28 (clk0, LatchA28, clk0, LatchB28);
Registre32b     lA29 (clk0, LatchA29, clk0, LatchB29);
Registre32b     lA30 (clk0, LatchA30, clk0, LatchB30);
Registre32b     lA31 (clk0, LatchA31, clk0, LatchB31);

//UPM function calls
//UPMR call
UPMR  c0 (InA2_c0, InA1_c0,InA0_c0, InX5_c0, InX2_c0, InX1_c0, InX0_c0, InX4_c0, InX3_c0, InX6_c0, InA4_c0,
        InA3_c0, InD2_c0, FP_clock, bit1, bit2, bit3_c0, ad0_c0, ad1_c0, ad2_c0, ad3_c0, ad4_c0,
        ad5_c0, Out0_c0, Out1_c0, Out4_c0, Out3_c0, Out2_c0, Out5_c0, Out6_c0, result2_c0);  //UPM call
//UPMR call
UPMR  c1 (InA2_c1, InA1_c1,InA0_c1, InX5_c1, InX2_c1, InX1_c1, InX0_c1, InX4_c1, InX3_c1, InX6_c1, InA4_c1,
        InA3_c1, InD2_c1, FP_clock, bit1, bit2, bit3_c1, ad0_c1, ad1_c1, ad2_c1, ad3_c1, ad4_c1,
        ad5_c1, Out0_c1, Out1_c1, Out4_c1, Out3_c1, Out2_c1, Out5_c1, Out6_c1, result2_c1);  //UPM call

UPMR  c2 (InA2_c2, InA1_c2,InA0_c2, InX5_c2, InX2_c2, InX1_c2, InX0_c2, InX4_c2, InX3_c2, InX6_c2, InA4_c2,
        InA3_c2, InD2_c2, FP_clock, bit1, bit2, bit3_c2, ad0_c2, ad1_c2, ad2_c2, ad3_c2, ad4_c2,
        ad5_c2, Out0_c2, Out1_c2, Out4_c2, Out3_c2, Out2_c2, Out5_c2, Out6_c2, result2_c2);  //UPM call
//UPMR call
UPMR  c3 (InA2_c3, InA1_c3,InA0_c3, InX5_c3, InX2_c3, InX1_c3, InX0_c3, InX4_c3, InX3_c3, InX6_c3, InA4_c3,
        InA3_c3, InD2_c3, FP_clock, bit1, bit2, bit3_c3, ad0_c3, ad1_c3, ad2_c3, ad3_c3, ad4_c3,
        ad5_c3, Out0_c3, Out1_c3, Out4_c3, Out3_c3, Out2_c3, Out5_c3, Out6_c3, result2_c3);  //UPM call
//UPMR call
UPMR  c4 (InA2_c4, InA1_c4,InA0_c4, InX5_c4, InX2_c4, InX1_c4, InX0_c4, InX4_c4, InX3_c4, InX6_c4, InA4_c4,
        InA3_c4, InD2_c4, FP_clock, bit1, bit2, bit3_c4, ad0_c4, ad1_c4, ad2_c4, ad3_c4, ad4_c4,
        ad5_c4, Out0_c4, Out1_c4, Out4_c4, Out3_c4, Out2_c4, Out5_c4, Out6_c4, result2_c4);  //UPM call
//UPMR call
UPMR  c5 (InA2_c5, InA1_c5,InA0_c5, InX5_c5, InX2_c5, InX1_c5, InX0_c5, InX4_c5, InX3_c5, InX6_c5, InA4_c5,
        InA3_c5, InD2_c5, FP_clock, bit1, bit2, bit3_c5, ad0_c5, ad1_c5, ad2_c5, ad3_c5, ad4_c5,
        ad5_c5, Out0_c5, Out1_c5, Out4_c5, Out3_c5, Out2_c5, Out5_c5, Out6_c5, result2_c5);  //UPM call
//UPMR call
UPMR  c6 (InA2_c6, InA1_c6,InA0_c6, InX5_c6, InX2_c6, InX1_c6, InX0_c6, InX4_c6, InX3_c6, InX6_c6, InA4_c6,
        InA3_c6, InD2_c6, FP_clock, bit1, bit2, bit3_c6, ad0_c6, ad1_c6, ad2_c6, ad3_c6, ad4_c6,
        ad5_c6, Out0_c6, Out1_c6, Out4_c6, Out3_c6, Out2_c6, Out5_c6, Out6_c6, result2_c6);  //UPM call
//UPMR call
UPMR  c7 (InA2_c7, InA1_c7,InA0_c7, InX5_c7, InX2_c7, InX1_c7, InX0_c7, InX4_c7, InX3_c7, InX6_c7, InA4_c7,
        InA3_c7, InD2_c7, FP_clock, bit1, bit2, bit3_c7, ad0_c7, ad1_c7, ad2_c7, ad3_c7, ad4_c7,
        ad5_c7, Out0_c7, Out1_c7, Out4_c7, Out3_c7, Out2_c7, Out5_c7, Out6_c7, result2_c7);  //UPM call
//UPMR call
UPMR  c8 (InA2_c8, InA1_c8,InA0_c8, InX5_c8, InX2_c8, InX1_c8, InX0_c8, InX4_c8, InX3_c8, InX6_c8, InA4_c8,
        InA3_c8, InD2_c8, FP_clock, bit1, bit2, bit3_c8, ad0_c8, ad1_c8, ad2_c8, ad3_c8, ad4_c8,
        ad5_c8, Out0_c8, Out1_c8, Out4_c8, Out3_c8, Out2_c8, Out5_c8, Out6_c8, result2_c8);  //UPM call
//UPMR call
```

ANNEXE 23 – Algorithme du processeur matriciel *6*x4 UPMs (5)

```
UPMR  c9 (InA2_c9, InA1_c9,InA0_c9, InX5_c9, InX2_c9, InX1_c9, InX0_c9, InX4_c9, InX3_c9, InX6_c9, InA4_c9,
        InA3_c9, InD2_c9, FP_clock, bit1, bit2, bit3_c9, ad0_c9, ad1_c9, ad2_c9, ad3_c9, ad4_c9,
        ad5_c9, Out0_c9, Out1_c9, Out4_c9, Out3_c9, Out2_c9, Out5_c9, Out6_c9, result2_c9);  //UPM call

//UPM call
UPM  c10 (InA2_c10, InA1_c10, InA0_c10, InX5_c10, InX2_c10, InX1_c10, InX0_c10, InX4_c10, InX3_c10, InA4_c10,
        InA3_c10, FP_clock, ad0_c10, ad1_c10, ad2_c10, ad3_c10, Out0_c10, Out1_c10, Out4_c10, Out2_c10);
//UPM call
UPM  c11 (InA2_c11, InA1_c11, InA0_c11, InX5_c11, InX2_c11, InX1_c11, InX0_c11, InX4_c11, InX3_c11,  InA4_c11,
        InA3_c11, FP_clock, ad0_c11, ad1_c11, ad2_c11, ad3_c11, Out0_c11, Out1_c11, Out4_c11, Out2_c11);


UPM  c12 (InA2_c12, InA1_c12, InA0_c12, InX5_c12, InX2_c12, InX1_c12, InX0_c12, InX4_c12, InX3_c12,  InA4_c12,
        InA3_c12, FP_clock, ad0_c12, ad1_c12, ad2_c12, ad3_c12, Out0_c12, Out1_c12, Out4_c12, Out2_c12);
//UPM call
UPM  c13 (InA2_c13, InA1_c13, InA0_c13, InX5_c13, InX2_c13, InX1_c13, InX0_c13, InX4_c13, InX3_c13,  InA4_c13,
        InA3_c13, FP_clock, ad0_c13, ad1_c13, ad2_c13, ad3_c13, Out0_c13, Out1_c13, Out4_c13, Out2_c13);
//UPM call
UPM  c14 (InA2_c14, InA1_c14, InA0_c14, InX5_c14, InX2_c14, InX1_c14, InX0_c14, InX4_c14, InX3_c14,  InA4_c14,
        InA3_c14, FP_clock, ad0_c14, ad1_c14, ad2_c14, ad3_c14, Out0_c14, Out1_c14, Out4_c14, Out2_c14);
//UPM call
UPM  c15 (InA2_c15, InA1_c15, InA0_c15, InX5_c15, InX2_c15, InX1_c15, InX0_c15, InX4_c15, InX3_c15,  InA4_c15,
        InA3_c15, FP_clock, ad0_c15, ad1_c15, ad2_c15, ad3_c15, Out0_c15, Out1_c15, Out4_c15, Out2_c15);
//UPM call
UPM  c16 (InA2_c16, InA1_c16, InA0_c16, InX5_c16, InX2_c16, InX1_c16, InX0_c16, InX4_c16, InX3_c16,  InA4_c16,
        InA3_c16, FP_clock, ad0_c16, ad1_c16, ad2_c16, ad3_c16, Out0_c16, Out1_c16, Out4_c16, Out2_c16);
//UPM call
UPM  c17 (InA2_c17, InA1_c17, InA0_c17, InX5_c17, InX2_c17, InX1_c17, InX0_c17, InX4_c17, InX3_c17,  InA4_c17,
        InA3_c17, FP_clock, ad0_c17, ad1_c17, ad2_c17, ad3_c17, Out0_c17, Out1_c17, Out4_c17, Out2_c17);
//UPM call
UPM  c18 (InA2_c18, InA1_c18, InA0_c18, InX5_c18, InX2_c18, InX1_c18, InX0_c18, InX4_c18, InX3_c18,  InA4_c18,
        InA3_c18, FP_clock, ad0_c18, ad1_c18, ad2_c18, ad3_c18, Out0_c18, Out1_c18, Out4_c18, Out2_c18);
//UPM call
UPM  c19 (InA2_c19, InA1_c19, InA0_c19, InX5_c19, InX2_c19, InX1_c19, InX0_c19, InX4_c19, InX3_c19,  InA4_c19,
        InA3_c19, FP_clock, ad0_c19, ad1_c19, ad2_c19, ad3_c19, Out0_c19, Out1_c19, Out4_c19, Out2_c19);
//UPM call
UPM  c20 (InA2_c20, InA1_c20, InA0_c20, InX5_c20, InX2_c20, InX1_c20, InX0_c20, InX4_c20, InX3_c20,  InA4_c20,
        InA3_c20, FP_clock, ad0_c20, ad1_c20, ad2_c20, ad3_c20, Out0_c20, Out1_c20, Out4_c20, Out2_c20);
//UPM call
UPM  c21 (InA2_c21, InA1_c21, InA0_c21, InX5_c21, InX2_c21, InX1_c21, InX0_c21, InX4_c21, InX3_c21,  InA4_c21,
        InA3_c21, FP_clock, ad0_c21, ad1_c21, ad2_c21, ad3_c21, Out0_c21, Out1_c21, Out4_c21, Out2_c21);
//UPM call
UPM  c22 (InA2_c22, InA1_c22, InA0_c22, InX5_c22, InX2_c22, InX1_c22, InX0_c22, InX4_c22, InX3_c22,  InA4_c22,
        InA3_c22, FP_clock, ad0_c22, ad1_c22, ad2_c22, ad3_c22, Out0_c22, Out1_c22, Out4_c22, Out2_c22);
//UPM call
UPM  c23 (InA2_c23, InA1_c23, InA0_c23, InX5_c23, InX2_c23, InX1_c23, InX0_c23, InX4_c23, InX3_c23,  InA4_c23,
        InA3_c23, FP_clock, ad0_c23, ad1_c23, ad2_c23, ad3_c23, Out0_c23, Out1_c23, Out4_c23, Out2_c23);

//Multiplexer 32x32bits call
Multiplexeur32x32b m1 (OutMux0, OutMux10, OutMux11, OutMux12, OutMux13, OutMux14, OutMux15, OutMux16,
            OutMux17, OutMux18, OutMux19, OutMux1, OutMux20, OutMux21, OutMux22, OutMux23, OutMux24,
            OutMux25, OutMux26, OutMux27, OutMux28, OutMux29, OutMux2, OutMux30, OutMux31, OutMux3,
            OutMux4, OutMux5, OutMux6, OutMux7, OutMux8, OutMux9, MuxCtrl, MuxOutput); //Multiplexer Function */

//Clock generators
delay         dl0   (FP_clock, 7'b1000001, S0);
clock_divider   cl0   (S0, 6'b001110, 6'b011010, clk_Opcode0);
delay         dl1   (clk_Opcode0, 7'b0000010, S1);
```

ANNEXE 24 – Algorithme du processeur matriciel 6x4 UPMs (6)

```
clock_divider   cl1   (S1, 6'b010001, 6'b100000, bit2_Opcode0);
clock_divider0      cl2   (FP_clock,   clk0_Opcode0);
clock_divider2      cl3   (FP_clock,   clk0_Opcode2);

//////////////////////FIR/IIR/////////FIR/////////All-zero////Racine///////all-pole////division/////fft///////
Multiplexeur8x1b   mx1 (clk_Opcode0, clk_Opcode0, clk_Opcode0, clk_Opcode0, clk_Opcode0, clk_Opcode0,clk_Opcode0,
clk_Opcode0, Opcode, clk);
Multiplexeur8x1b     mx2   (bit2_Opcode0,   clk_Opcode0,   bit2_Opcode0,   clk_Opcode0,        clk_Opcode0,   bit2_Opcode0,
bit2_Opcode0, bit2_Opcode0, Opcode, bit2);
Multiplexeur8x1b     mx3   (clk0_Opcode0,   clk0_Opcode0,   clk0_Opcode0,   clk0_Opcode0,   clk0_Opcode0,   clk0_Opcode0,
clk0_Opcode0, clk0_Opcode0, Opcode, clk0);

//Device Timing Algorithm
always @ (posedge clk)
begin /*
//////////////////////////////
/////ADAPTIVE FIR-IIR//////N=20///
//////////////////////////////
 if (Opcode==3'b000)
begin
//SEMI-ADAPTIVE FIR-IIR
// processor input
   InA2_c0 <= Samples;
//Control bits
bit1       <= 1'b0;
bit3_c0 <= 1'b1;
bit3_c1 <= 1'b1;
bit3_c2 <= 1'b1;
bit3_c3 <= 1'b1;
bit3_c4 <= 1'b1;
bit3_c5 <= 1'b1;
bit3_c6 <= 1'b1;
bit3_c7 <= 1'b1;
bit3_c8 <= 1'b1;
bit3_c9 <= 1'b1;
   ad0_c0  <= 1'b1;
   ad1_c0  <= 1'b1;
   ad2_c0  <= 1'b1;
   ad3_c0  <= 1'b1;
   ad4_c0  <= 1'b1;
   ad5_c0  <= 1'b0;
   ad0_c1  <= 1'b1;
   ad1_c1  <= 1'b1;
   ad2_c1  <= 1'b1;
   ad3_c1  <= 1'b1;
   ad4_c1  <= 1'b1;
   ad5_c1  <= 1'b0;
   ad0_c2  <= 1'b1;
   ad1_c2  <= 1'b1;
   ad2_c2  <= 1'b1;
   ad3_c2  <= 1'b1;
   ad4_c2  <= 1'b1;
   ad5_c2  <= 1'b0;
   ad0_c3  <= 1'b1;
   ad1_c3  <= 1'b1;
   ad2_c3  <= 1'b1;
   ad3_c3  <= 1'b1;
   ad4_c3  <= 1'b1;
   ad5_c3  <= 1'b0;
   ad0_c4  <= 1'b1;
   ad1_c4  <= 1'b1;
   ad2_c4  <= 1'b1;
   ad3_c4  <= 1'b1;
   ad4_c4  <= 1'b1;
   ad5_c4  <= 1'b0;
   ad0_c5  <= 1'b1;
   ad1_c5  <= 1'b1;
   ad2_c5  <= 1'b1;
   ad3_c5  <= 1'b1;
   ad4_c5  <= 1'b1;
   ad5_c5  <= 1'b0;
   ad0_c6  <= 1'b1;
   ad1_c6  <= 1'b1;
   ad2_c6  <= 1'b1;
   ad3_c6  <= 1'b1;
   ad4_c6  <= 1'b1;
   ad5_c6  <= 1'b0;
   ad0_c7  <= 1'b1;
   ad1_c7  <= 1'b1;
   ad2_c7  <= 1'b1;
   ad3_c7  <= 1'b1;
   ad4_c7  <= 1'b1;
   ad5_c7  <= 1'b0;
   ad0_c8  <= 1'b1;
   ad1_c8  <= 1'b1;
   ad2_c8  <= 1'b1;
   ad3_c8  <= 1'b1;
   ad4_c8  <= 1'b1;
   ad5_c8  <= 1'b0;
   ad0_c9  <= 1'b1;
   ad1_c9  <= 1'b1;
   ad2_c9  <= 1'b1;
   ad3_c9  <= 1'b1;
   ad4_c9  <= 1'b1;
   ad5_c9  <= 1'b0;
   //Initialization of
   //UPM c0 parameters
InA1_c0  <= zero;
   InA0_c0 <= zero;
   InX5_c0 <= zero;
   InX2_c0 <= data0;
   InX1_c0 <= data1;
   InX0_c0 <= data2;
   InX4_c0 <= zero;
   InX3_c0 <= zero;
   InA4_c0 <= un;
   InA3_c0 <= un;
   InD2_c0 <= zero;
   //Interconnection
   //with neighbourhood
   InX6_c0 <= Out3_c9;
   InX5_c1 <= Out1_c0;
   InD2_c1 <= Out4_c0;
   InX6_c1 <= Out6_c0;
   //Initialization
   //of UPM c1
InA2_c1 <= result2_c0;
InA1_c1  <= zero;
   InA0_c1 <= zero;
   InX2_c1 <= data3;
   InX1_c1 <= data4;
   InX0_c1 <= data5;
   InX4_c1 <= zero;
   InX3_c1 <= zero;
   InA4_c1 <= un;
   InA3_c1 <= un;
   //Interconnection
   //with neighbourhood
   InX5_c2 <= Out1_c1;
   InD2_c2 <= Out4_c1;
   InX6_c2 <= Out6_c1;
    //Initialization
   //of UPM c2 parameters
InA2_c2 <= zero;
InA1_c2  <= zero;
   InA0_c2 <= zero;
   InX2_c2 <= zero;
   InX1_c2 <= zero;
   InX0_c2 <= zero;
   InX4_c2 <= zero;
   InX3_c2 <= zero;
   InA4_c2 <= un;
   InA3_c2 <= un;
     //Interconnection
   //with neighbourhood
   InX5_c3 <= Out1_c2;
   InD2_c3 <= Out4_c2;
   InX6_c3 <= Out6_c2;
   //Initialization of
   //UPM c3 parameters
InA2_c3 <= result2_c1;
InA1_c3  <= zero;
   InA0_c3 <= zero;
   InX2_c3 <= data6;
   InX1_c3 <= data7;
   InX0_c3 <= data8;
   InX4_c3 <= zero;
   InX3_c3 <= zero;
   InA4_c3 <= un;
   InA3_c3 <= un;
   //Interconnection
   //with neighbourhood
   InX5_c4 <= Out1_c3;
   InD2_c4 <= Out4_c3;
   InX6_c4 <= Out6_c3;
    //Initialization of
   //UPM c4 parameters
InA2_c4 <= result2_c3;
InA1_c4  <= zero;
   InA0_c4 <= zero;
   InX2_c4 <= data9;
   InX1_c4 <= data10;
   InX0_c4 <= data11;
   InX4_c4 <= zero;
   InX3_c4 <= zero;
   InA4_c4 <= un;
   InA3_c4 <= un;
   //Interconnection
   //with neighbourhood
   InX5_c5 <= Out1_c4;
   InD2_c5 <= Out4_c4;
   InX6_c5 <= Out6_c4;
    //Initialization
   //of UPM c5
   InA2_c5 <= zero;
InA1_c5  <= zero;
   InA0_c5 <= zero;
   InX2_c5 <= zero;
   InX1_c5 <= zero;
   InX0_c5 <= zero;
   InX4_c5 <= zero;
```

ANNEXE 25 – Algorithme du processeur matriciel 6x4 UPMs (7)

```
InX3_c5 <= zero;
   InA4_c5 <= un;
   InA3_c5 <= un;
    //Interconnection
   //with neighbourhood
   InX5_c6 <= Out1_c5;
   InD2_c6 <= Out4_c5;
   InX6_c6 <= Out6_c5;
    //Initialization of
   //UPM c6 parameters
InA2_c6 <= result2_c4;
  InA1_c6          <=
zero;
   InA0_c6 <= zero;
   InX2_c6 <= data12;
   InX1_c6 <= data13;
   InX0_c6 <= data14;
   InX4_c6 <= zero;
   InX3_c6 <= zero;
   InA4_c6 <= un;
   InA3_c6 <= un;
    //Interconnection
   //with neighbourhood
UPM
   InX5_c7 <= Out1_c6;
   InD2_c7 <= Out4_c6;
   InX6_c7 <= Out6_c6;
    //Initialization of
   //UPM c7 parameters
InA2_c7 <= result2_c6;
InA1_c7  <= zero;
   InA0_c7 <= zero;
   InX2_c7 <= data15;
   InX1_c7 <= data16;
   InX0_c7 <= data17;
   InX4_c7 <= zero;
   InX3_c7 <= zero;
   InA4_c7 <= un;
   InA3_c7 <= un;
   //Interconnection
   //with neighbourhood
   InX5_c8 <= Out1_c7;
   InD2_c8 <= Out4_c7;
   InX6_c8 <= Out6_c7;
    //Initialization
   //of UPM c8
   InA2_c8 <= zero;
InA1_c8  <= zero;
   InA0_c8 <= zero;
   InX2_c8 <= zero;
   InX1_c8 <= zero;
   InX0_c8 <= zero;
   InX4_c8 <= zero;
   InX3_c8 <= zero;
   InA4_c8 <= un;
   InA3_c8 <= un;
   //Interconnection
   //with neighbourhood
   InX5_c9 <= Out1_c8;
   InD2_c9 <= Out4_c8;
   InX6_c9 <= Out6_c8;
   //Initialization of
   //UPM c9 parameters
InA2_c9 <= result2_c7;
InA1_c9  <= zero;
   InA0_c9 <= zero;
   InX2_c9 <= data18;
   InX1_c9 <= data19;
   InX0_c9 <= data20;
   InX4_c9 <= zero;
   InX3_c9 <= zero;
   InA4_c9 <= un;
   InA3_c9 <= un;
   //FIR/IIR
   //processor output
   Output0 <= Out1_c9;
   Output1 <= Out3_c9;
end
////////////////////////////
/////// ADAPTIVE FIR
///////N=30///////
else                    if
(Opcode==3'b001)
          begin
InA2_c0 <= Samples;
//Control bits
bit1       <= 1'b0;
bit3_c0 <= 1'b1;
bit3_c1 <= 1'b1;
bit3_c2 <= 1'b1;
bit3_c3 <= 1'b1;
bit3_c4 <= 1'b1;
bit3_c5 <= 1'b1;
bit3_c6 <= 1'b1;
bit3_c7 <= 1'b1;
bit3_c8 <= 1'b1;
bit3_c9 <= 1'b1;
//Add/sub bits
   ad0_c0  <= 1'b1;
   ad1_c0  <= 1'b1;
   ad2_c0  <= 1'b1;
   ad3_c0  <= 1'b1;
   ad4_c0  <= 1'b1;
   ad5_c0  <= 1'b0;
   ad0_c1  <= 1'b1;
   ad1_c1  <= 1'b1;
   ad2_c1  <= 1'b1;
   ad3_c1  <= 1'b1;
   ad4_c1  <= 1'b1;
   ad5_c1  <= 1'b0;
   ad0_c2  <= 1'b1;
   ad1_c2  <= 1'b1;
   ad2_c2  <= 1'b1;
   ad3_c2  <= 1'b1;
   ad4_c2  <= 1'b1;
   ad5_c2  <= 1'b0;
   ad0_c3  <= 1'b1;
   ad1_c3  <= 1'b1;
   ad2_c3  <= 1'b1;
   ad3_c3  <= 1'b1;
   ad4_c3  <= 1'b1;
   ad5_c3  <= 1'b0;
   ad0_c4  <= 1'b1;
   ad1_c4  <= 1'b1;
   ad2_c4  <= 1'b1;
   ad3_c4  <= 1'b1;
   ad4_c4  <= 1'b1;
   ad5_c4  <= 1'b0;
   ad0_c5  <= 1'b1;
   ad1_c5  <= 1'b1;
   ad2_c5  <= 1'b1;
   ad3_c5  <= 1'b1;
   ad4_c5  <= 1'b1;
   ad5_c5  <= 1'b0;
   ad0_c6  <= 1'b1;
   ad1_c6  <= 1'b1;
   ad2_c6  <= 1'b1;
   ad3_c6  <= 1'b1;
   ad4_c6  <= 1'b1;
   ad5_c6  <= 1'b0;
   ad0_c7  <= 1'b1;
   ad1_c7  <= 1'b1;
   ad2_c7  <= 1'b1;
   ad3_c7  <= 1'b1;
   ad4_c7  <= 1'b1;
   ad5_c7  <= 1'b0;
   ad0_c8  <= 1'b1;
   ad1_c8  <= 1'b1;
   ad2_c8  <= 1'b1;
   ad3_c8  <= 1'b1;
   ad4_c8  <= 1'b1;
   ad5_c8  <= 1'b0;
   ad0_c9  <= 1'b1;
   ad1_c9  <= 1'b1;
   ad2_c9  <= 1'b1;
   ad3_c9  <= 1'b1;
   ad4_c9  <= 1'b1;
   ad5_c9  <= 1'b0;
   //Initialization of
   //UPM c0 parameters
InA1_c0  <= zero;
   InA0_c0 <= zero;
   InX5_c0 <= zero;
   InX2_c0 <= data0;
   InX1_c0 <= data1;
   InX0_c0 <= data2;
   InX4_c0 <= zero;
   InX3_c0 <= zero;
   InA4_c0 <= zero;
   InA3_c0 <= zero;
   InD2_c0 <= zero;
   //Interconnection
   //with neighbourhood
   InX6_c0 <= Out3_c9;
   InX5_c1 <= Out1_c0;
   InD2_c1 <= Out4_c0;
   InX6_c1 <= Out6_c0;
   //Initialization
   //of UPM c1
InA2_c1 <= result2_c0;
InA1_c1  <= zero;
   InA0_c1 <= zero;
   InX2_c1 <= data3;
   InX1_c1 <= data4;
   InX0_c1 <= data5;
   InX4_c1 <= zero;
   InX3_c1 <= zero;
   InA4_c1 <= zero;
   InA3_c1 <= zero;
   //Interconnection
   //with neighbourhood
   InX5_c2 <= Out1_c1;
   InD2_c2 <= Out4_c1;
   InX6_c2 <= Out6_c1;
   //Initialization
   //of UPM c2
 InA2_c2 <= result2_c1;
InA1_c2  <= zero;
   InA0_c2 <= zero;
   InX2_c2 <= data6;
   InX1_c2 <= data7;
   InX0_c2 <= data8;
   InX4_c2 <= zero;
   InX3_c2 <= zero;
   InA4_c2 <= zero;
   InA3_c2 <= zero;
   //Interconnection
   //with neighbourhood
   InX5_c3 <= Out1_c2;
   InD2_c3 <= Out4_c2;
   InX6_c3 <= Out6_c2;
   //Initialization of
   //UPM c3 parameters
   InA2_c3          <=
result2_c2;
   InA1_c3          <=
zero;
   InA0_c3 <= zero;
   InX2_c3 <= data9;
   InX1_c3 <= data10;
   InX0_c3 <= data11;
   InX4_c3 <= zero;
   InX3_c3 <= zero;
   InA4_c3 <= zero;
   InA3_c3 <= zero;
   //Interconnection
```

ANNEXE 26 – Algorithme du processeur matriciel *6*x4 UPMs (8)

```
  //with    neighbourhood
UPM
    InX5_c4 <= Out1_c3;
    InD2_c4 <= Out4_c3;
    InX6_c4 <= Out6_c3;
     //Initialization of
    //UPM c4 parameters
InA2_c4 <= result2_c3;
InA1_c4  <= zero;
    InA0_c4 <= zero;
    InX2_c4 <= data12;
    InX1_c4 <= data13;
    InX0_c4 <= data14;
    InX4_c4 <= zero;
    InX3_c4 <= zero;
    InA4_c4 <= zero;
    InA3_c4 <= zero;
    //Interconnection
    //with neighbourhood
    InX5_c5 <= Out1_c4;
    InD2_c5 <= Out4_c4;
    InX6_c5 <= Out6_c4;
    //Initialization
    //of UPM c5
InA2_c5 <= result2_c4;
InA1_c5  <= zero;
    InA0_c5 <= zero;
    InX2_c5 <= data15;
    InX1_c5 <= data16;
    InX0_c5 <= data17;
    InX4_c5 <= zero;
    InX3_c5 <= zero;
    InA4_c5 <= zero;
    InA3_c5 <= zero;
    //Interconnection
    //with neighbourhood
    InX5_c6 <= Out1_c5;
    InD2_c6 <= Out4_c5;
    InX6_c6 <= Out6_c5;
    //Initialization of
    //UPM c6 parameters
InA2_c6 <= result2_c5;
    InA1_c6          <=
zero;
    InA0_c6 <= zero;
    InX2_c6 <= data18;
    InX1_c6 <= data19;
    InX0_c6 <= data20;
    InX4_c6 <= zero;
    InX3_c6 <= zero;
    InA4_c6 <= zero;
    InA3_c6 <= zero;
    //Interconnection
    //with neighbourhood
    InX5_c7 <= Out1_c6;
    InD2_c7 <= Out4_c6;
    InX6_c7 <= Out6_c6;
    //Initialization of
    //UPM c7 parameters
    InA2_c7 <= result2_c6;
    InA1_c7  <= zero;
       InA0_c7 <= zero;
       InX2_c7 <= data20;
       InX1_c7 <= data21;
       InX0_c7 <= data22;
       InX4_c7 <= zero;
       InX3_c7 <= zero;
       InA4_c7 <= zero;
       InA3_c7 <= zero;
       //Interconnection
       //with neighbourhood
       InX5_c8 <= Out1_c7;
       InD2_c8 <= Out4_c7;
       InX6_c8 <= Out6_c7;
       //Initialization
       //of UPM c8
    InA2_c8 <= result2_c7;
    InA1_c8  <= zero;
       InA0_c8 <= zero;
       InX2_c8 <= data23;
       InX1_c8 <= data24;
       InX0_c8 <= data25;
       InX4_c8 <= zero;
       InX3_c8 <= zero;
       InA4_c8 <= zero;
       InA3_c8 <= zero;
       //Interconnection
       //with neighbourhood
       InX5_c9 <= Out1_c8;
       InD2_c9 <= Out4_c8;
       InX6_c9 <= Out6_c8;
       //Initialization of
       //UPM c9 parameters
    InA2_c9 <= result2_c8;
    InA1_c9  <= zero;
       InA0_c9 <= zero;
       InX2_c9 <= data26;
       InX1_c9 <= data27;
       InX0_c9 <= data28;
       InX4_c9 <= zero;
       InX3_c9 <= zero;
       InA4_c9 <= zero;
       InA3_c9 <= zero;
       //FIR Correlator
       Output0 <= Out1_c9;
       Output1 <= Out3_c9;
       end
    /////ALL-POLE
    LATTICE MODE///
    ////////////////////////////
    else  if
    (Opcode==3'b010)
    begin
     //ALL-POLE
    //Processor input
    InX2_c0 <= Samples;
    //Control bits
    bit1 <= 1'b1;
    bit3_c0 <= 1'b1;
    bit3_c1 <= 1'b1;
    bit3_c2 <= 1'b1;
    bit3_c3 <= 1'b1;
    bit3_c4 <= 1'b1;
    bit3_c5 <= 1'b1;
    bit3_c6 <= 1'b1;
    bit3_c7 <= 1'b1;
    bit3_c8 <= 1'b1;
    bit3_c9 <= 1'b1;
    //Add/Sub bits
       ad0_c0  <= 1'b1;
       ad1_c0  <= 1'b0;
       ad2_c0  <= 1'b1;
       ad3_c0  <= 1'b1;
       ad4_c0  <= 1'b1;
       ad5_c0  <= 1'b0;
       ad0_c1  <= 1'b1;
       ad1_c1  <= 1'b0;
       ad2_c1  <= 1'b1;
       ad3_c1  <= 1'b1;
       ad4_c1  <= 1'b1;
       ad5_c1  <= 1'b0;
       ad0_c2  <= 1'b1;
       ad1_c2  <= 1'b0;
       ad2_c2  <= 1'b1;
       ad3_c2  <= 1'b1;
       ad4_c2  <= 1'b1;
       ad5_c2  <= 1'b0;
       ad0_c3  <= 1'b1;
       ad1_c3  <= 1'b0;
       ad2_c3  <= 1'b1;
       ad3_c3  <= 1'b1;
       ad4_c3  <= 1'b1;
       ad5_c3  <= 1'b0;
       ad0_c4  <= 1'b1;
       ad1_c4  <= 1'b0;
       ad2_c4  <= 1'b1;
       ad3_c4  <= 1'b1;
       ad4_c4  <= 1'b1;
       ad5_c4  <= 1'b0;
       ad0_c5  <= 1'b1;
       ad1_c5  <= 1'b0;
       ad2_c5  <= 1'b1;
       ad3_c5  <= 1'b1;
       ad4_c5  <= 1'b1;
       ad5_c5  <= 1'b1;
       ad0_c6  <= 1'b1;
       ad1_c6  <= 1'b0;
       ad2_c6  <= 1'b1;
       ad3_c6  <= 1'b1;
       ad4_c6  <= 1'b1;
       ad5_c6  <= 1'b0;
       ad0_c7  <= 1'b1;
       ad1_c7  <= 1'b0;
       ad2_c7  <= 1'b1;
       ad3_c7  <= 1'b1;
       ad4_c7  <= 1'b1;
       ad5_c7  <= 1'b0;
       ad0_c8  <= 1'b1;
       ad1_c8  <= 1'b0;
       ad2_c8  <= 1'b1;
       ad3_c8  <= 1'b1;
       ad4_c8  <= 1'b1;
       ad5_c8  <= 1'b0;
       ad0_c9  <= 1'b1;
       ad1_c9  <= 1'b0;
       ad2_c9  <= 1'b1;
       ad3_c9  <= 1'b1;
       ad4_c9  <= 1'b1;
       ad5_c9  <= 1'b0;
       //Initialization of
       //UPM c0 parameters
    InA2_c0 <= un;
    InA1_c0  <= un;
       InA0_c0 <= zero;
       InX5_c0 <= zero;
       InX0_c0 <= zero;
       InA4_c0 <= un;
       InA3_c0 <= un;
       InD2_c0 <= zero;
       //Interconnection
       //with neighbourhood
       InX1_c0 <= Out5_c0;
       InX4_c0 <= Out5_c0;
       InX3_c0 <= Out0_c0;
       InX6_c0 <= Out4_c1;
       //Next UPM input
       InX2_c1 <= Out0_c0;
       //Initialization of
       //UPM c1 parameters
    InA2_c1 <= un;
    InA1_c1  <= un;
       InA0_c1 <= zero;
       InX5_c1 <= zero;
       InX0_c1 <= zero;
       InA4_c1 <= un;
       InA3_c1 <= un;
       InD2_c1 <= zero;
        //Interconnection
       //with neighbourhood
       InX1_c1 <= Out5_c1;
       InX4_c1 <= Out5_c1;
       InX3_c1 <= Out0_c1;
       InX6_c1 <= Out4_c2;
       //Next UPM input
       InX2_c2 <= Out0_c1;
       //Initialization of
       //UPM c2 parameters
    InA2_c2 <= un;
    InA1_c2  <= un;
    InA0_c2 <= zero;
       InX5_c2 <= zero;
       InX0_c2 <= zero;
```

ANNEXE 27 – Algorithme du processeur matriciel *6*x4 UPMs (9)

```
 InA4_c2 <= un;
    InA3_c2 <= un;
    InD2_c2 <= zero;
     //Interconnection
    //with neighbourhood
    InX1_c2 <= Out5_c2;
    InX4_c2 <= Out5_c2;
    InX3_c2 <= Out0_c2;
    InX6_c2 <= Out4_c3;
    //Next UPM input
    InX2_c3 <= Out0_c2;
    //Initialization of
    //UPM c3 parameters
 InA2_c3 <= un;
 InA1_c3  <= un;
    InA0_c3 <= zero;
    InX5_c3 <= zero;
    InX0_c3 <= zero;
    InA4_c3 <= un;
    InA3_c3 <= un;
    InD2_c3 <= zero;
 //Interconnection
    //with neighbourhood
    InX1_c3 <= Out5_c3;
    InX4_c3 <= Out5_c3;
    InX3_c3 <= Out0_c3;
    InX6_c3 <= Out4_c4;
    //Next UPM input
    InX2_c4 <= Out0_c3;
    //Initialization of
    //UPM c4 parameters
 InA2_c4 <= un;
 InA1_c4  <= un;
    InA0_c4 <= zero;
    InX5_c4 <= zero;
    InX0_c4 <= zero;
    InA4_c4 <= un;
    InA3_c4 <= un;
    InD2_c4 <= zero;
 //Interconnection
 //with neighbourhood
    InX1_c4 <= Out5_c4;
    InX4_c4 <= Out5_c4;
    InX3_c4 <= Out0_c4;
    InX6_c4 <= Out4_c5;
    //Next UPM input
    InX2_c5 <= Out0_c4;
    //Initialization of
    //UPM c5 parameters
 InA2_c5 <= un;
 InA1_c5  <= un;
    InA0_c5 <= zero;
    InX5_c5 <= zero;
    InX0_c5 <= zero;
    InA4_c5 <= un;
    InA3_c5 <= un;
    InD2_c5 <= zero;
 //Interconnection
  //with neighbourhood
    InX1_c5 <= Out5_c5;
    InX4_c5 <= Out5_c5;
    InX3_c5 <= Out0_c5;
    InX6_c5 <= Out4_c6;
     //Next UPM input
    InX2_c6 <= Out0_c5;
    //Initialization of
    //UPM c2 parameters
 InA2_c6 <= un;
 InA1_c6  <= un;
    InA0_c6 <= zero;
    InX5_c6 <= zero;
    InX0_c6 <= zero;
    InA4_c6 <= un;
    InA3_c6 <= un;
    InD2_c6 <= zero;
 //Interconnection
 //with neighbourhood
    InX1_c6 <= Out5_c6;
    InX4_c6 <= Out5_c6;
    InX3_c6 <= Out0_c6;
    InX6_c6 <= Out4_c7;
     //Next UPM input
    InX2_c7 <= Out0_c6;
    //Initialization of
    //UPM c2 parameters
 InA2_c7 <= un;
 InA1_c7  <= un;
    InA0_c7 <= zero;
    InX5_c7 <= zero;
    InX0_c7 <= zero;
    InA4_c7 <= un;
    InA3_c7 <= un;
    InD2_c7 <= zero;
 //Interconnection with
 //neighbourhood inputs
    InX1_c7 <= Out5_c7;
    InX4_c7 <= Out5_c7;
    InX3_c7 <= Out0_c7;
    InX6_c7 <= Out4_c8;
     //Next UPM input
    InX2_c8 <= Out0_c7;
    //Initialization of
    //UPM c2 parameters
 InA2_c8 <= un;
 InA1_c8  <= un;
    InA0_c8 <= zero;
    InX5_c8 <= zero;
    InX0_c8 <= zero;
    InA4_c8 <= un;
    InA3_c8 <= un;
    InD2_c8 <= zero;
 //Interconnection with
  //neighbourhood inputs
    InX1_c8 <= Out5_c8;
    InX4_c8 <= Out5_c8;
    InX3_c8 <= Out0_c8;
    InX6_c8 <= Out4_c9;
    //Next UPM input
    InX2_c9 <= Out0_c8;
    //Initialization of
    //UPM c2 parameters
 InA2_c9 <= un;
 InA1_c9  <= un;
    InA0_c9 <= zero;
    InX5_c9 <= zero;
    InX0_c9 <= zero;
    InA4_c9 <= un;
    InA3_c9 <= un;
    InD2_c9 <= zero;
 //Interconnection
    //with neighborhood
    InX1_c9 <= Out5_c9;
    InX4_c9 <= Out5_c9;
    InX3_c9 <= Out0_c9;
    InX6_c9 <= Out0_c9;
    //ALL-POLE
    //processor output
    Output0 <= Out0_c9;
    Output1 <= Out4_c9;
 end
 ////////////////////////////
 ///////RACINE
 CARRÉE////////
 ////////////////////////////
 else if
  (Opcode==3'b011)
 begin
 ///1 ere DIVISION
 //Control bits
 bit1        <= 1'b1;
 bit3_c0 <= 1'b1;
 bit3_c1 <= 1'b1;
 bit3_c2 <= 1'b1;
 bit3_c3 <= 1'b1;
 bit3_c4 <= 1'b1;
 bit3_c5 <= 1'b1;
 bit3_c6 <= 1'b1;
 bit3_c7 <= 1'b1;
 bit3_c8 <= 1'b1;
 bit3_c9 <= 1'b1;
             /////////////////
    ad0_c0  <= 1'b1;
    ad1_c0  <= 1'b0;
    ad2_c0  <= 1'b1;
    ad3_c0  <= 1'b1;
    ad4_c0  <= 1'b1;
    ad5_c0  <= 1'b0;
    ad0_c1  <= 1'b1;
    ad1_c1  <= 1'b0;
    ad2_c1  <= 1'b1;
    ad3_c1  <= 1'b1;
    ad4_c1  <= 1'b1;
    ad5_c1  <= 1'b0;
    ad0_c2  <= 1'b1;
    ad1_c2  <= 1'b0;
    ad2_c2  <= 1'b1;
    ad3_c2  <= 1'b1;
    ad4_c2  <= 1'b1;
    ad5_c2  <= 1'b0;
    ad0_c3  <= 1'b1;
    ad1_c3  <= 1'b0;
    ad2_c3  <= 1'b1;
    ad3_c3  <= 1'b1;
    ad4_c3  <= 1'b1;
    ad5_c3  <= 1'b0;
    ////////////
    ad0_c4  <= 1'b1;
    ad1_c4  <= 1'b0;
    ad2_c4  <= 1'b1;
    ad3_c4  <= 1'b1;
    ad4_c4  <= 1'b1;
    ad5_c4  <= 1'b1;
 ///////////
    ad0_c5  <= 1'b1;
    ad1_c5  <= 1'b0;
    ad2_c5  <= 1'b1;
    ad3_c5  <= 1'b1;
    ad4_c5  <= 1'b1;
    ad5_c5  <= 1'b0;
    ad0_c6  <= 1'b1;
    ad1_c6  <= 1'b0;
    ad2_c6  <= 1'b1;
    ad3_c6  <= 1'b1;
    ad4_c6  <= 1'b1;
    ad5_c6  <= 1'b0;
    ad0_c7  <= 1'b1;
    ad1_c7  <= 1'b0;
    ad2_c7  <= 1'b1;
    ad3_c7  <= 1'b1;
    ad4_c7  <= 1'b1;
    ad5_c7  <= 1'b0;
    ad0_c8  <= 1'b1;
    ad1_c8  <= 1'b0;
    ad2_c8  <= 1'b1;
    ad3_c8  <= 1'b1;
    ad4_c8  <= 1'b1;
    ad5_c8  <= 1'b0;
    ad0_c9  <= 1'b1;
    ad1_c9  <= 1'b0;
    ad2_c9  <= 1'b1;
    ad3_c9  <= 1'b1;
    ad4_c9  <= 1'b1;
    ad5_c9  <= 1'b1;
    ad0_c10  <= 1'b1;
    ad1_c10  <= 1'b0;
    ad2_c10  <= 1'b1;
    ad3_c10  <= 1'b1;
    ad0_c11  <= 1'b1;
    ad1_c11  <= 1'b0;
    ad2_c11  <= 1'b1;
```

ANNEXE 28 – Algorithme du processeur matriciel *6*x4 UPMs (10)

```
 ad3_c11  <= 1'b1;
    //Initialization of
    //UPM c0 parameters
  //Initialization of
    //UPM c0 parameters
InA2_c0 <= data2;
InA1_c0  <= data0;
    InA0_c0 <= data2;
    InX5_c0 <= zero;
    InX2_c0 <= two;
    InX1_c0 <= Out2_c0;
    InX0_c0 <= data2;
    InA4_c0 <= zero;
    InA3_c0 <= zero;
    InD2_c0 <= zero;
    InX4_c0 <= zero;
    InX3_c0 <= zero;
    InX6_c0 <= zero;
    //Initialization of
    //UPM c1 parameters
InA2_c1 <= Out0_c0;
InA1_c1  <= data0;
    InA0_c1 <= Out0_c0;
    InX5_c1 <= zero;
    InX2_c1 <= two;
    InX1_c1 <= Out2_c1;
    InX0_c1 <= Out0_c0;
    InA4_c1 <= zero;
    InA3_c1 <= zero;
    InD2_c1 <= zero;
    InX4_c1 <= zero;
    InX3_c1 <= zero;
    InX6_c1 <= zero;
    //Initialization of
    //UPM c2 parameters
InA2_c2 <= Out0_c1;
InA1_c2  <= data0;
    InA0_c2 <= Out0_c1;
    InX5_c2 <= zero;
    InX2_c2 <= two;
    InX1_c2 <= Out2_c2;
    InX0_c2 <= Out0_c1;
    InA4_c2 <= zero;
    InA3_c2 <= zero;
    InD2_c2 <= zero;
    InX4_c2 <= zero;
    InX3_c2 <= zero;
    InX6_c2 <= zero;
    //DIVISION
    //processor output
    InA2_c3 <= data2;
InA1_c3  <= Samples;
InA0_c3                 <=
zero_p_cinq;
    InX5_c3 <= zero;
    InX2_c3 <= un;
    InX1_c3 <= Out0_c2;
    InX0_c3 <= Out0_c3;
    InA4_c3 <= zero;
    InA3_c3 <= zero;
    InD2_c3 <= zero;
    InX4_c3 <= zero;
    InX3_c3 <= zero;
    InX6_c3 <= zero;
       //Initialization of
    //UPM c2 parameters
InA2_c4 <= Out0_c2;
InA1_c4  <= Out2_c3;
    InA0_c4 <= Out0_c2;
    InX5_c4 <= zero;
    InX2_c4 <= two;
    InX1_c4 <= Out2_c4;
    InX0_c4 <= Out0_c2;
    InA4_c4 <= zero;
    InA3_c4 <= zero;
    InD2_c4 <= zero;
    InX4_c4 <= zero;
    InX3_c4 <= zero;
    InX6_c4 <= zero;
      //UPM              c0
parameters
InA2_c5 <= Out0_c4;
InA1_c5  <= Out2_c3;
    InA0_c5 <= Out0_c4;
    InX5_c5 <= zero;
    InX2_c5 <= two;
    InX1_c5 <= Out2_c5;
    InX0_c5 <= Out0_c4;
    InA4_c5 <= zero;
    InA3_c5 <= zero;
    InD2_c5 <= zero;
    InX4_c5 <= zero;
    InX3_c5 <= zero;
    InX6_c5 <= zero;
    //Initialization of
    //UPM c1 parameters
InA2_c6 <= Out0_c5;
InA1_c6  <= Out2_c3;
    InA0_c6 <= Out0_c5;
    InX5_c6 <= zero;
    InX2_c6 <= two;
    InX1_c6 <= Out2_c6;
    InX0_c6 <= Out0_c5;
    InA4_c6 <= zero;
    InA3_c6 <= zero;
    InD2_c6 <= zero;
    InX4_c6 <= zero;
    InX3_c6 <= zero;
    InX6_c6 <= zero;
     //DIVISION
    //processor output
    InA2_c7 <= Out2_c3;
InA1_c7  <= Samples;
InA0_c7                 <=
zero_p_cinq;
    InX5_c7 <= zero;
    InX2_c7 <= un;
    InX1_c7 <= Out0_c6;
    InX0_c7 <= Out0_c7;
    InA4_c7 <= zero;
    InA3_c7 <= zero;
    InD2_c7 <= zero;
    InX4_c7 <= zero;
    InX3_c7 <= zero;
    InX6_c7 <= zero;
    //Initialization of
    //UPM c2 parameters
InA2_c8 <= Out0_c6;
InA1_c8  <= Out2_c7;
    InA0_c8 <= Out0_c6;
    InX5_c8 <= zero;
    InX2_c8 <= two;
    InX1_c8 <= Out2_c8;
    InX0_c8 <= Out0_c6;
    InA4_c8 <= zero;
    InA3_c8 <= zero;
    InD2_c8 <= zero;
    InX4_c8 <= zero;
    InX3_c8 <= zero;
    InX6_c8 <= zero;
       //Initialization of
    //UPM c2 parameters
InA2_c9 <= Out0_c8;
InA1_c9  <= Out2_c7;
    InA0_c9 <= Out0_c8;
    InX5_c9 <= zero;
    InX2_c9 <= two;
    InX1_c9 <= Out2_c9;
    InX0_c9 <= Out0_c8;
    InA4_c9 <= zero;
    InA3_c9 <= zero;
    InD2_c9 <= zero;
    InX4_c9 <= zero;
    InX3_c9 <= zero;
    InX6_c9 <= zero;
    //Initialization of
    //UPM c2 parameters
InA2_c10 <= Out0_c9;
InA1_c10 <= Out2_c9;
InA0_c10 <= Out0_c9;
InX5_c10 <= zero;
InX2_c10 <= two;
InX1_c10 <= Out2_c10;
InX0_c10 <= Out0_c9;
    InA4_c10 <= zero;
    InA3_c10 <= zero;
    InD2_c10 <= zero;
    InX4_c10 <= zero;
    InX3_c10 <= zero;
    InX6_c10 <= zero;
    //Initialization of
    //UPM c2 parameters
     //DIVISION
    //processor output
InA2_c11 <= Out2_c7;
InA1_c11 <= Samples;
InA0_c11                <=
zero_p_cinq;
    InX5_c11 <= zero;
    InX2_c11 <= un;
InX1_c11 <= Out0_c10;
InX0_c11 <= Out0_c11;
    InA4_c11 <= zero;
    InA3_c11 <= zero;
    InD2_c11 <= zero;
    InX4_c11 <= zero;
    InX3_c11 <= zero;
    InX6_c11 <= zero;
    //Square
    //processor output
Output0 <= Out2_c11;
Output1 <= Out0_c10;
end
//////////////////////////////
/////ALL-ZERO
LATTICE MODE///
//////////////////////////////
 else  if
(Opcode==3'b100)
begin
     //Processor inputs
    InX2_c0 <= Samples;
LatchA0  <= Samples;
    InX1_c0 <= data1x;
    InX4_c0 <= data1x;
    InX3_c0 <= Samples;
//Control bits
bit1 <= 1'b1;
 bit3_c0 <= 1'b1;
  bit3_c1 <= 1'b1;
bit3_c2 <= 1'b1;
bit3_c3 <= 1'b1;
bit3_c4 <= 1'b1;
bit3_c5 <= 1'b1;
bit3_c6 <= 1'b1;
bit3_c7 <= 1'b1;
bit3_c8 <= 1'b1;
bit3_c9 <= 1'b1;
//Add/Sub bits
    ad0_c0  <= 1'b1;
    ad1_c0  <= 1'b1;
    ad2_c0  <= 1'b1;
    ad3_c0  <= 1'b1;
```

ANNEXE 29 – Algorithme du processeur matriciel *6*x4 UPMs (11)

```
 ad4_c0  <= 1'b1;
   ad5_c0  <= 1'b1;
   ad0_c1  <= 1'b1;
   ad1_c1  <= 1'b1;
   ad2_c1  <= 1'b1;
   ad3_c1  <= 1'b1;
   ad4_c1  <= 1'b1;
   ad5_c1  <= 1'b1;
   ad0_c2  <= 1'b1;
   ad1_c2  <= 1'b1;
   ad2_c2  <= 1'b1;
   ad3_c2  <= 1'b1;
   ad4_c2  <= 1'b1;
   ad5_c2  <= 1'b1;
   ad0_c3  <= 1'b1;
   ad1_c3  <= 1'b1;
   ad2_c3  <= 1'b1;
   ad3_c3  <= 1'b1;
   ad4_c3  <= 1'b1;
   ad5_c3  <= 1'b1;
   ad0_c4  <= 1'b1;
   ad1_c4  <= 1'b1;
   ad2_c4  <= 1'b1;
   ad3_c4  <= 1'b1;
   ad4_c4  <= 1'b1;
   ad5_c4  <= 1'b1;
   ad0_c5  <= 1'b1;
   ad1_c5  <= 1'b1;
   ad2_c5  <= 1'b1;
   ad3_c5  <= 1'b1;
   ad4_c5  <= 1'b1;
   ad5_c5  <= 1'b1;
   ad0_c6  <= 1'b1;
   ad1_c6  <= 1'b1;
   ad2_c6  <= 1'b1;
   ad3_c6  <= 1'b1;
   ad4_c6  <= 1'b1;
   ad5_c6  <= 1'b1;
   ad0_c7  <= 1'b1;
   ad1_c7  <= 1'b1;
   ad2_c7  <= 1'b1;
   ad3_c7  <= 1'b1;
   ad4_c7  <= 1'b1;
   ad5_c7  <= 1'b1;
   ad0_c8  <= 1'b1;
   ad1_c8  <= 1'b1;
   ad2_c8  <= 1'b1;
   ad3_c8  <= 1'b1;
   ad4_c8  <= 1'b1;
   ad5_c8  <= 1'b1;
   ad0_c9  <= 1'b1;
   ad1_c9  <= 1'b1;
   ad2_c9  <= 1'b1;
   ad3_c9  <= 1'b1;
   ad4_c9  <= 1'b1;
   ad5_c9  <= 1'b1;
   //Initialization of
   //UPM c0 parameters
InA2_c0 <= un;
InA1_c0  <= un;
   InA0_c0 <= zero;
   InX5_c0 <= zero;
   InX0_c0 <= zero;
   InA4_c0 <= un;
   InA3_c0 <= un;
   InD2_c0 <= zero;
   //Interconnection
   //with neighbourhood
   InX6_c0 <= Out4_c0;
   //Next UPM inputs
   InX2_c1 <= Out0_c0;
   InX3_c1 <= Out0_c0;
   InX1_c1 <= Out5_c0;
   InX4_c1 <= Out5_c0;
   //UPM c1 parameters
InA2_c1 <= un;
InA1_c1  <= un;
   InA0_c1 <= zero;
   InX5_c1 <= zero;
   InX0_c1 <= zero;
   InA4_c1 <= un;
   InA3_c1 <= un;
   InD2_c1 <= zero;
    //Interconnection
   //with neighbourhood
   InX6_c1 <= Out4_c1;
    //Next UPM inputs
   InX2_c2 <= Out0_c1;
   InX3_c2 <= Out0_c1;
   InX1_c2 <= Out5_c1;
   InX4_c2 <= Out5_c1;
    //Initialization of
   //UPM c2 parameters
InA2_c2 <= un;
InA1_c2  <= un;
   InA0_c2 <= zero;
   InX5_c2 <= zero;
   InX0_c2 <= zero;
   InA4_c2 <= un;
   InA3_c2 <= un;
   InD2_c2 <= zero;
    //Interconnection
   //with  neighbourhood
inputs
   InX6_c2 <= Out4_c2;
   //Next UPM inputs
   InX2_c3 <= Out0_c2;
   InX3_c3 <= Out0_c2;
   InX1_c3 <= Out5_c2;
   InX4_c3 <= Out5_c2;
   //UPM c3 parameters
InA2_c3 <= un;
InA1_c3  <= un;
   InA0_c3 <= zero;
   InX5_c3 <= zero;
   InX0_c3 <= zero;
   InA4_c3 <= un;
   InA3_c3 <= un;
   InD2_c3 <= zero;
    //Interconnection
   //with neighbourhood
   InX6_c3 <= Out4_c3;
   //Next UPM inputs
   InX2_c4 <= Out0_c3;
   InX3_c4 <= Out0_c3;
   InX1_c4 <= Out5_c3;
   InX4_c4 <= Out5_c3;
    //Initialization of
   //UPM c4 parameters
InA2_c4 <= un;
InA1_c4  <= un;
   InA0_c4 <= zero;
   InX5_c4 <= zero;
   InX0_c4 <= zero;
   InA4_c4 <= un;
   InA3_c4 <= un;
   InD2_c4 <= zero;
   //with neighbourhood
   InX6_c4 <= Out4_c4;
   //Next UPM inputs
   InX2_c5 <= Out0_c4;
   InX3_c5 <= Out0_c4;
   InX1_c5 <= Out5_c4;
   InX4_c5 <= Out5_c4;
   //UPM c5 parameters
InA2_c5 <= un;
InA1_c5  <= un;
   InA0_c5 <= zero;
   InX5_c5 <= zero;
   InX0_c5 <= zero;
   InA4_c5 <= un;
   InA3_c5 <= un;
   InD2_c5 <= zero;
    //Interconnection
   //with neighbourhood
   InX6_c5 <= Out4_c5;
   //Next UPM inputs
   InX2_c6 <= Out0_c5;
   InX3_c6 <= Out0_c5;
   InX1_c6 <= Out5_c5;
   InX4_c6 <= Out5_c5;
   //UPM c6 parameters
InA2_c6 <= un;
InA1_c6  <= un;
   InA0_c6 <= zero;
   InX5_c6 <= zero;
   InX0_c6 <= zero;
   InA4_c6 <= un;
   InA3_c6 <= un;
   InD2_c6 <= zero;
    //Interconnection
   //with neighbourhood
   InX6_c6 <= Out4_c6;
   //Next UPM inputs
   InX2_c7 <= Out0_c6;
   InX3_c7 <= Out0_c6;
   InX1_c7 <= Out5_c6;
   InX4_c7 <= Out5_c6;
   //UPM c7 parameters
InA2_c7 <= un;
InA1_c7  <= un;
   InA0_c7 <= zero;
   InX5_c7 <= zero;
   InX0_c7 <= zero;
   InA4_c7 <= un;
   InA3_c7 <= un;
   InD2_c7 <= zero;
    //Interconnection
   //with neighbourhood
   InX6_c7 <= Out4_c7;
   //Next UPM inputs
   InX2_c8 <= Out0_c7;
   InX3_c8 <= Out0_c7;
   InX1_c8 <= Out5_c7;
   InX4_c8 <= Out5_c7;
   //UPM c8 parameters
InA2_c8 <= un;
InA1_c8  <= un;
   InA0_c8 <= zero;
   InX5_c8 <= zero;
   InX0_c8 <= zero;
   InA4_c8 <= un;
   InA3_c8 <= un;
   InD2_c8 <= zero;
    //Interconnection
   //with neighbourhood
   InX6_c8 <= Out4_c8;
   //Next UPM inputs
   InX2_c9 <= Out0_c8;
   InX3_c9 <= Out0_c8;
   InX1_c9 <= Out5_c8;
   InX4_c9 <= Out5_c8;
   //UPM c5 parameters
InA2_c9 <= un;
InA1_c9  <= un;
   InA0_c9 <= zero;
   InX5_c9 <= zero;
   InX0_c9 <= zero;
   InA4_c9 <= un;
   InA3_c9 <= un;
   InD2_c9 <= zero;
   //with neighbourhood
   InX6_c9 <= Out4_c9;
    //ALL-ZERO
LATTICE outputs
   Output0 <= Out0_c9;
   Output1 <= Out4_c9;
   end

   ////////////////////////
/////DiIVISION////////////
////////////////////////
else                    if
(Opcode==3'b101)
begin
//Control bits
```

ANNEXE 30 – Algorithme du processeur matriciel *6*x4 UPMs (12)

```
bit1 <= 1'b1;
bit3_c0 <= 1'b1;
bit3_c1 <= 1'b1;
bit3_c2 <= 1'b1;
bit3_c3 <= 1'b1;
bit3_c4 <= 1'b1;
bit3_c5 <= 1'b1;
bit3_c6 <= 1'b1;
bit3_c7 <= 1'b1;
bit3_c8 <= 1'b1;
bit3_c9 <= 1'b1;
//Add/Sub bits
   ad0_c0  <= 1'b1;
   ad1_c0  <= 1'b0;
   ad2_c0  <= 1'b1;
   ad3_c0  <= 1'b1;
   ad4_c0  <= 1'b1;
   ad5_c0  <= 1'b0;
   ad0_c1  <= 1'b1;
   ad1_c1  <= 1'b0;
   ad2_c1  <= 1'b1;
   ad3_c1  <= 1'b1;
   ad4_c1  <= 1'b1;
   ad5_c1  <= 1'b0;
   ad0_c2  <= 1'b1;
   ad1_c2  <= 1'b0;
   ad2_c2  <= 1'b1;
   ad3_c2  <= 1'b1;
   ad4_c2  <= 1'b1;
   ad5_c2  <= 1'b0;
   ad0_c3  <= 1'b1;
   ad1_c3  <= 1'b0;
   ad2_c3  <= 1'b1;
   ad3_c3  <= 1'b1;
   ad4_c3  <= 1'b1;
   ad5_c3  <= 1'b0;
   ad0_c4  <= 1'b1;
   ad1_c4  <= 1'b0;
   ad2_c4  <= 1'b1;
   ad3_c4  <= 1'b1;
   ad4_c4  <= 1'b1;
   ad5_c4  <= 1'b0;
   ad0_c5  <= 1'b1;
   ad1_c5  <= 1'b0;
   ad2_c5  <= 1'b1;
   ad3_c5  <= 1'b1;
   ad4_c5  <= 1'b1;
   ad5_c5  <= 1'b0;
   ad0_c6  <= 1'b1;
   ad1_c6  <= 1'b0;
   ad2_c6  <= 1'b1;
   ad3_c6  <= 1'b1;
   ad4_c6  <= 1'b1;
   ad5_c6  <= 1'b0;
   ad0_c7  <= 1'b1;
   ad1_c7  <= 1'b0;
   ad2_c7  <= 1'b1;
   ad3_c7  <= 1'b1;
   ad4_c7  <= 1'b1;
   ad5_c7  <= 1'b0;
   ad0_c8  <= 1'b1;
   ad1_c8  <= 1'b0;
   ad2_c8  <= 1'b1;
   ad3_c8  <= 1'b1;
   ad4_c8  <= 1'b1;
   ad5_c8  <= 1'b0;
   ad0_c9  <= 1'b1;
   ad1_c9  <= 1'b0;
   ad2_c9  <= 1'b1;
   ad3_c9  <= 1'b1;
   ad4_c9  <= 1'b1;
   ad5_c9  <= 1'b0;
   //Initialization of
   //UPM c0 parameters
InA2_c0 <= Samples1;
InA1_c0  <= Samples;
InA0_c0 <= Samples1;
   InX5_c0 <= zero;
   InX2_c0 <= two;
   InX1_c0 <= Out2_c0;
InX0_c0 <= Samples1;
   InA4_c0 <= zero;
   InA3_c0 <= zero;
   InD2_c0 <= zero;
   InX4_c0 <= zero;
   InX3_c0 <= zero;
   InX6_c0 <= zero;
   //Initialization of
   //UPM c1 parameters
InA2_c1 <= Out0_c0;
InA1_c1  <= Samples;
   InA0_c1 <= Out0_c0;
   InX5_c1 <= zero;
   InX2_c1 <= two;
   InX1_c1 <= Out2_c1;
   InX0_c1 <= Out0_c0;
   InA4_c1 <= zero;
   InA3_c1 <= zero;
   InD2_c1 <= zero;
   InX4_c1 <= zero;
   InX3_c1 <= zero;
   InX6_c1 <= zero;
   //Initialization of
   //UPM c2 parameters
InA2_c2 <= Out0_c1;
InA1_c2  <= Samples;
   InA0_c2 <= Out0_c1;
   InX5_c2 <= zero;
   InX2_c2 <= two;
   InX1_c2 <= Out2_c2;
   InX0_c2 <= Out0_c1;
   InA4_c2 <= zero;
   InA3_c2 <= zero;
   InD2_c2 <= zero;
   InX4_c2 <= zero;
   InX3_c2 <= zero;
   InX6_c2 <= zero;
   //Initialization of
   //UPM c2 parameters
InA2_c3 <= Out0_c2;
InA1_c3  <= Samples;
   InA0_c3 <= Out0_c2;
   InX5_c3 <= zero;
   InX2_c3 <= two;
   InX1_c3 <= Out2_c3;
   InX0_c3 <= Out0_c2;
   InA4_c3 <= zero;
   InA3_c3 <= zero;
   InD2_c3 <= zero;
   InX4_c3 <= zero;
   InX3_c3 <= zero;
   InX6_c3 <= zero;
   //UPM c4 parameters
InA2_c4 <= Out0_c3;
InA1_c4  <= Samples;
   InA0_c4 <= Out0_c3;
   InX5_c4 <= zero;
   InX2_c4 <= two;
   InX1_c4 <= Out2_c4;
   InX0_c4 <= Out0_c3;
   InA4_c4 <= zero;
   InA3_c4 <= zero;
   InD2_c4 <= zero;
   InX4_c4 <= zero;
   InX3_c4 <= zero;
   InX6_c4 <= zero;
   //UPM c5 parameters
InA2_c5 <= Out0_c4;
InA1_c5  <= Samples;
   InA0_c5 <= Out0_c4;
   InX5_c5 <= zero;
   InX2_c5 <= two;
   InX1_c5 <= Out2_c5;
   InX0_c5 <= Out0_c4;
   InA4_c5 <= zero;
   InA3_c5 <= zero;
   InD2_c5 <= zero;
   InX4_c5 <= zero;
   InX3_c5 <= zero;
   InX6_c5 <= zero;
   //UPM c6 parameters
InA2_c6 <= Out0_c5;
InA1_c6  <= Samples;
   InA0_c6 <= Out0_c5;
   InX5_c6 <= zero;
   InX2_c6 <= two;
   InX1_c6 <= Out2_c6;
   InX0_c6 <= Out0_c5;
   InA4_c6 <= zero;
   InA3_c6 <= zero;
   InD2_c6 <= zero;
   InX4_c6 <= zero;
   InX3_c6 <= zero;
   InX6_c6 <= zero;
   //UPM c7 parameters
InA2_c7 <= Out0_c6;
InA1_c7  <= Samples;
   InA0_c7 <= Out0_c6;
   InX5_c7 <= zero;
   InX2_c7 <= two;
   InX1_c7 <= Out2_c5;
   InX0_c7 <= Out0_c6;
   InA4_c7 <= zero;
   InA3_c7 <= zero;
   InD2_c7 <= zero;
   InX4_c7 <= zero;
   InX3_c7 <= zero;
   InX6_c7 <= zero;
   //UPM c8 parameters
InA2_c8 <= Out0_c7;
InA1_c8  <= Samples;
   InA0_c8 <= Out0_c7;
   InX5_c8 <= zero;
   InX2_c8 <= two;
   InX1_c8 <= Out2_c8;
   InX0_c8 <= Out0_c7;
   InA4_c8 <= zero;
   InA3_c8 <= zero;
   InD2_c8 <= zero;
   InX4_c8 <= zero;
   InX3_c8 <= zero;
   InX6_c8 <= zero;
   //UPM c9 parameters
InA2_c9 <= Out0_c8;
InA1_c9  <= Samples;
   InA0_c9 <= Out0_c8;
   InX5_c9 <= zero;
   InX2_c9 <= two;
   InX1_c9 <= Out2_c9;
   InX0_c9 <= Out0_c8;
   InA4_c9 <= zero;
   InA3_c9 <= zero;
   InD2_c9 <= zero;
   InX4_c9 <= zero;
   InX3_c9 <= zero;
   InX6_c9 <= zero;
   //DIVISION
   //processor output
   Output0 <= Out0_c9;
   Output1 <= zero;
end
//////////////////////////////
/////Radix-4  FFT  N=16
(OIOO)//
//////////////////////////////
 else  if
(Opcode==3'b101)
begin
if (bit2==1'b0)
begin
//////////////////////
```

ANNEXE 31 – Algorithme du processeur matriciel *6*x4 UPMs (13)

```
//First array of four//
//Radix-4 Butterflies//
//Processor input
//Control bits
//Configuration
bit1 <= 1'b1;
//
bit3_c0 <= 1'b0;
bit3_c1 <= 1'b0;
bit3_c2 <= 1'b0;
bit3_c3 <= 1'b0;
bit3_c4 <= 1'b0;
bit3_c5 <= 1'b0;
bit3_c6 <= 1'b0;
bit3_c7 <= 1'b0;
bit3_c8 <= 1'b0;
bit3_c9 <= 1'b0;
//C0 add_sub bits
   ad0_c0  <= 1'b1;
   ad1_c0  <= 1'b1;
   ad2_c0  <= 1'b0;
   ad3_c0  <= 1'b1;
   ad4_c0  <= 1'b1;
   ad5_c0  <= 1'b1;
   ////C1 add_sub bit
   ad0_c1  <= 1'b1;
   ad1_c1  <= 1'b1;
   ad2_c1  <= 1'b0;
   ad3_c1  <= 1'b1;
   ad4_c1  <= 1'b1;
   ad5_c1  <= 1'b1;
   //C2 add_sub bit
   ad0_c2  <= 1'b0;
   ad1_c2  <= 1'b1;
   ad2_c2  <= 1'b0;
   ad3_c2  <= 1'b0;
   ad4_c2  <= 1'b1;
   ad5_c2  <= 1'b1;
   //C3 add_sub bit
   ad0_c3  <= 1'b0;
   ad1_c3  <= 1'b1;
   ad2_c3  <= 1'b0;
   ad3_c3  <= 1'b0;
   ad4_c3  <= 1'b1;
   ad5_c3  <= 1'b1;
   //C4 add_sub bits
   ad0_c4  <= 1'b1;
   ad1_c4  <= 1'b1;
   ad2_c4  <= 1'b0;
   ad3_c4  <= 1'b1;
   ad4_c4  <= 1'b1;
   ad5_c4  <= 1'b1;
   ////C5 add_sub bit
   ad0_c5  <= 1'b1;
   ad1_c5  <= 1'b1;
   ad2_c5  <= 1'b0;
   ad3_c5  <= 1'b1;
   ad4_c5  <= 1'b1;
   ad5_c5  <= 1'b1;
   //C6 add_sub bit
   ad0_c6  <= 1'b0;
   ad1_c6  <= 1'b1;
   ad2_c6  <= 1'b0;
   ad3_c6  <= 1'b0;
   ad4_c6  <= 1'b1;
   ad5_c6  <= 1'b1;
   //C7 add_sub bit
   ad0_c7  <= 1'b0;
   ad1_c7  <= 1'b1;
   ad2_c7  <= 1'b0;
   ad3_c7  <= 1'b0;
   ad4_c7  <= 1'b1;
   ad5_c7  <= 1'b1;
   //C8 add_sub bits
   ad0_c8  <= 1'b1;
   ad1_c8  <= 1'b1;
   ad2_c8  <= 1'b0;
   ad3_c8  <= 1'b1;
   ad4_c8  <= 1'b1;
   ad5_c8  <= 1'b1;
   ////C9 add_sub bit
   ad0_c9  <= 1'b1;
   ad1_c9  <= 1'b1;
   ad2_c9  <= 1'b0;
   ad3_c9  <= 1'b1;
   ad4_c9  <= 1'b1;
   ad5_c9  <= 1'b1;
   //C10 add_sub bit
   ad0_c10  <= 1'b0;
   ad1_c10  <= 1'b1;
   ad2_c10  <= 1'b0;
   ad3_c10  <= 1'b0;
   //C11 add_sub bit
   ad0_c11  <= 1'b0;
   ad1_c11  <= 1'b1;
   ad2_c11  <= 1'b0;
   ad3_c11  <= 1'b0;
    //C12 add_sub bits
   ad0_c12  <= 1'b1;
   ad1_c12  <= 1'b1;
   ad2_c12  <= 1'b0;
   ad3_c12  <= 1'b1;
   ////C13 add_sub bit
   ad0_c13  <= 1'b1;
   ad1_c13  <= 1'b1;
   ad2_c13  <= 1'b0;
   ad3_c13  <= 1'b1;
   //C14 add_sub bit
   ad0_c14  <= 1'b0;
   ad1_c14  <= 1'b1;
   ad2_c14  <= 1'b0;
   ad3_c14  <= 1'b0;
   //C15 add_sub bit
   ad0_c15  <= 1'b0;
   ad1_c15  <= 1'b1;
   ad2_c15  <= 1'b0;
   ad3_c15  <= 1'b0;
   //C16 add_sub bits
ad0_c16  <= 1'b1;
   ad1_c16  <= 1'b0;
   ad2_c16  <= 1'b1;
   ad3_c16  <= 1'b1;
   ////C17 add_sub bit
   ad0_c17  <= 1'b1;
   ad1_c17  <= 1'b0;
   ad2_c17  <= 1'b1;
   ad3_c17  <= 1'b1;
   //C18 add_sub bit
   ad0_c18  <= 1'b1;
   ad1_c18  <= 1'b0;
   ad2_c18  <= 1'b1;
   ad3_c18  <= 1'b1;
   //C19 add_sub bit
   ad0_c19  <= 1'b1;
   ad1_c19  <= 1'b0;
   ad2_c19  <= 1'b1;
   ad3_c19  <= 1'b0;
   //C20 add_sub bit
   ad0_c20  <= 1'b1;
   ad1_c20  <= 1'b0;
   ad2_c20  <= 1'b1;
   ad3_c20  <= 1'b1;
   //C21 add_sub bit
   ad0_c21  <= 1'b1;
   ad1_c21  <= 1'b0;
   ad2_c21  <= 1'b1;
   ad3_c21  <= 1'b1;
   //C22 add_sub bit
   ad0_c22  <= 1'b1;
   ad1_c22  <= 1'b0;
   ad2_c22  <= 1'b1;
   ad3_c22  <= 1'b1;
   //C23 add_sub bits
   ad0_c23  <= 1'b1;
   ad1_c23  <= 1'b0;
   ad2_c23  <= 1'b1;
   ad3_c23  <= 1'b1;
////////////////////
//Butterfly              r=4
configurations
   //Butterfly A//
   //Configuration of
   // 2 UPMs c0 and c1
   InA2_c0 <= un;
InA1_c0  <= un;
InA0_c0 <= two;
InX5_c0    <=    data32;
//R(f0)
   InX2_c0   <=   data28;
//R(f2)
   InX1_c0 <= Out4_c0;
//R(f3+f1)
   InX0_c0 <= Out4_c0;
//R(f3+f1)
   InX4_c0   <=   data30;
//R(f1)
   InX3_c0   <=   data26;
//R(f3)
   InA4_c0 <= un;
   InA3_c0 <= un;
   InD2_c0 <= zero;
   InX6_c0 <= zero;
   //
   InA2_c1 <= un;
InA1_c1  <= un;
InA0_c1 <= two;
 InX5_c1    <=    data31;
//I(f0)
   InX2_c1   <=   data27;
//I(f2)
   InX1_c1 <= Out4_c1;
//I(f3+f1)
   InX0_c1 <= Out4_c1;
//I(f3+f1)
   InX4_c1   <=   data29;
//I(f1)
   InX3_c1   <=   data25;
//I(f3)
   InA4_c1 <= un;
   InA3_c1 <= un;
   InD2_c1 <= zero;
   InX6_c1 <= zero;
   //Configuration of
   // 2 UPMs c2 and c3
InA2_c2 <= un;
InA1_c2  <= moinsun;
InA0_c2 <= moinstwo;
   InX5_c2   <=   data32;
//R(f0)
   InX2_c2   <=   data28;
//R(f2)
   InX1_c2 <= Out4_c3;
//I(f1-f3)
   InX0_c2 <= Out4_c3;
//I(f1-f3)
   InX4_c2   <=   data30;
//R(f1)
   InX3_c2   <=   data26;
//R(f3)
   InA4_c2 <= un;
   InA3_c2 <= un;
   InD2_c2 <= zero;
   InX6_c2 <= zero;
   //
   InA2_c3 <= un;
InA1_c3  <= un;
InA0_c3 <= two;
InX5_c3    <=    data31;
//I(f0)
   InX2_c3   <=   data27;
//I(f2)
```

ANNEXE 32 – Algorithme du processeur matriciel *6*x4 UPMs (14)

```
  InX1_c3 <= Out4_c2; //R(f1-f3)
    InX0_c3 <= Out4_c2; //R(f1-f3)
    InX4_c3 <= data29; //I(f1)
    InX3_c3 <= data25; //I(f3)
    InA4_c3 <= un;
    InA3_c3 <= un;
    InD2_c3 <= zero;
    InX6_c3 <= zero;
    //r=4 Butterfly A
    //Out0_c0=R(A0)
    //Out0_c1=I(A0)
    //Out1_c2=R(A1)
    //Out1_c3=I(A1)
    //Out1_c0=R(A2)
    //Out1_c1=I(A2)
    //Out0_c2=R(A3)
    //Out0_c3=I(A3)
//////////////////////////
      //Butterfly B//
     //Configuration of
    // 2 UPMs c4 and c5
    InA2_c4 <= un;
InA1_c4 <= un;
    InA0_c4 <= two;
    InX5_c4 <= data24; //R(f4)
    InX2_c4 <= data20; //R(f6)
    InX1_c4 <= Out4_c4; //R(f7+f5)
    InX0_c4 <= Out4_c4; //R(f7+f5)
    InX4_c4 <= data22; //R(f5)
    InX3_c4 <= data18; //R(f7)
    InA4_c4 <= un;
    InA3_c4 <= un;
    InD2_c4 <= zero;
    InX6_c4 <= zero;
    //
    InA2_c5 <= un;
InA1_c5 <= un;
    InA0_c5 <= two;
    InX5_c5 <= data23; //I(f4)
    InX2_c5 <= data19; //I(f6)
    InX1_c5 <= Out4_c5; //I(f7+f5)
    InX0_c5 <= Out4_c5; //I(f7+f5)
    InX4_c5 <= data21; //I(f5)
    InX3_c5 <= data17; //I(f7)
    InA4_c5 <= un;
    InA3_c5 <= un;
    InD2_c5 <= zero;
    InX6_c5 <= zero;
   //Configuration of
    // 2 UPMs c6 and c7
InA2_c6 <= un;
InA1_c6 <= moinsun;
InA0_c6 <= moinstwo;
    InX5_c6 <= data24; //R(f4)
    InX2_c6 <= data20; //R(f6)
    InX1_c6 <= Out4_c7; //I(f5-f7)
    InX0_c6 <= Out4_c7; //I(f5-f7)
    InX4_c6 <= data30; //R(f1)
    InX3_c6 <= data26; //R(f3)
    InA4_c6 <= un;
    InA3_c6 <= un;
    InD2_c6 <= zero;
    InX6_c6 <= zero;
    //
InA2_c7 <= un;
InA1_c7 <= un;
    InA0_c7 <= two;
    InX5_c7 <= data23; //I(f4)
    InX2_c7 <= data19; //I(f6)
    InX1_c7 <= Out4_c6; //R(f5-f7)
    InX0_c7 <= Out4_c6; //R(f5-f7)
    InX4_c7 <= data21; //I(f5)
    InX3_c7 <= data17; //I(f7)
    InA4_c7 <= un;
    InA3_c7 <= un;
    InD2_c7 <= zero;
    InX6_c7 <= zero;
    //r=4 Butterfly B
    //Out0_c4=R(B0)
    //Out0_c5=I(B0)
    //Out1_c6=R(B1)
    //Out1_c7=I(B1)
    //Out1_c4=R(B2)
    //Out1_c5=I(B2)
    //Out0_c6=R(B3)
    //Out0_c7=I(B3)
//////////////////////
    //Butterfly C//
     //Configuration of
    // 2 UPMs c8 and c9
InA2_c8 <= un;
InA1_c8 <= un;
    InA0_c8 <= two;
    InX5_c8 <= data16; //R(f8)
    InX2_c8 <= data12; //R(f10)
    InX1_c8 <= Out4_c8; //R(f11+f9)
    InX0_c8 <= Out4_c8; //R(f11+f9)
    InX4_c8 <= data14; //R(f9)
    InX3_c8 <= data9; //R(f11)
    InA4_c8 <= un;
    InA3_c8 <= un;
    InD2_c8 <= zero;
    InX6_c8 <= zero;
    //
InA2_c9 <= un;
InA1_c9 <= un;
    InA0_c9 <= two;
    InX5_c9 <= data15; //I(f8)
    InX2_c9 <= data11; //I(f10)
    InX1_c9 <= Out4_c9; //I(f11+f9)
    InX0_c9 <= Out4_c9; //I(f11+f9)
    InX4_c9 <= data13; //I(f9)
    InX3_c9 <= data9; //I(f11)
    InA4_c9 <= un;
    InA3_c9 <= un;
    InD2_c9 <= zero;
    InX6_c9 <= zero;
    //Configuration of
    // 2 UPMs c10 and
    InA2_c10 <= un;
InA1_c10 <= moinsun;
InA0_c10 <= moinstwo;
    InX5_c10 <= data16; //R(f8)
    InX2_c10 <= data12; //R(f10)
InX1_c10 <= Out4_c11; //I(f9-f11)
    InX0_c10 <= Out4_c11; //I(f9-f11)
    InX4_c10 <= data14; //R(f9)
    InX3_c10 <= data10; //R(f11)
    InA4_c10 <= un;
    InA3_c10 <= un;
    //
    InA2_c11 <= un;
    InA1_c11 <= un;
    InA0_c11 <= two;
    InX5_c11 <= data15; //I(f8)
    InX2_c11 <= data11; //I(f10)
    InX1_c11 <= Out4_c10; //R(f9-f11)
    InX0_c11 <= Out4_c10; //R(f9-f11)
    InX4_c11 <= data13; //I(f9)
    InX3_c11 <= data9; //I(f11)
    InA4_c11 <= un;
    InA3_c11 <= un;
    //r=4 Butterfly C
    //Out0_c8=R(C0)
    //Out0_c9=I(C0)
    //Out1_c10=R(C1)
    //Out1_c11=I(C1)
    //Out1_c8=R(C2)
    //Out1_c9=I(C2)
    //Out0_c10=R(C3)
    //Out0_c11=I(C3)
////////////////////
    //Butterfly D//
     //Configuration of
    // 2 UPMs c12 and
    InA2_c12 <= un;
    InA1_c12 <= un;
    InA0_c12 <= two;
    InX5_c12 <= data8; //R(f12)
    InX2_c12 <= data4; //R(f14)
InX1_c12 <= Out4_c12; //R(f15+f13)
InX0_c12 <= Out4_c12; //R(f15+f13)
InX4_c12 <= data6; //R(f13)
InX3_c12 <= data2; //R(f15)
    InA4_c12 <= un;
    InA3_c12 <= un;
    //
    InA2_c13 <= un;
    InA1_c13 <= un;
    InA0_c13 <= two;
InX5_c13 <= data7; //I(f12)
    InX2_c13 <= data3; //I(f14)
InX1_c13 <= Out4_c13 ; //I(f15+f13)
```

ANNEXE 33 – Algorithme du processeur matriciel *6*x4 UPMs (15)

```
InX0_c13 <= Out4_c13 ; //I(f15+f13)
   InX4_c13 <= data5; //I(f13)
   InX3_c13 <= data1;//I(f15)
   InA4_c13 <= un;
   InA3_c13 <= un;
    //Configuration of
   // 2 UPMs c14 and
InA2_c14 <= un;
InA1_c14 <= moinsun;
InA0_c14 <= moinstwo;
   InX5_c14 <= data8; //R(f12)
   InX2_c14 <= data4; //R(f14)
   InX1_c14 <= Out4_c15; //I(f13-f15)
   InX0_c14 <= Out4_c15; //I(f13-f15)
   InX4_c14 <= data6; //R(f13)
   InX3_c14 <= data2; //R(f15)
   InA4_c14 <= un;
   InA3_c14 <= un;
   //
   InA2_c15 <= un;
   InA1_c15 <= un;
   InA0_c15 <= two;
   InX5_c15 <= data7; //I(f12)
   InX2_c15 <= data3;//I(f14)
   InX1_c15 <= Out4_c14;//R(f13-f15)
   InX0_c15 <= Out4_c14;//R(f13-f15)
   InX4_c15 <= data5; //I(f13)
   InX3_c15 <= data1;//I(f15)
   InA4_c15 <= un;
   InA3_c15 <= un;
   //r=4 Butterfly D
   //Out0_c12=R(D0)
   //Out0_c13=I(D0)
   //Out1_c14=R(D1)
   //Out1_c15=I(D1)
   //Out1_c12=R(D2)
    //Out1_c13=I(D2)
   //Out0_c14=R(D3)
   //Out0_c15=I(D3)
////////////////////////////////
//Twidle factor
//Configuration c16
InA2_c16 <= Out1_c6; //R(B1)
InA1_c16 <= Out1_c7; //I(B1)
   InA0_c16 <= zero;
   InX5_c16 <= zero;
   InX2_c16 <= RW1; //R(W1)
   InX1_c16 <= IW1; //I(W1)
   InX0_c16 <= zero;
   InX4_c16 <= Out1_c7; //I(B1)
   InX3_c16 <= Out1_c6; //R(B1)
   InA4_c16 <= RW1; //R(W1)
   InA3_c16 <= IW1; //I(W1)
   //Configuration of c17
   InA2_c17 <= Out1_c10; //R(C1)
   InA1_c17 <= Out1_c11; //I(C1)
   InA0_c17 <= zero;
   InX5_c17 <= zero;
   InX2_c17 <= RW2; //R(W2)
   InX1_c17 <= IW2; //I(W2)
   InX0_c17 <= zero;
   InX4_c17 <= Out1_c11; //I(C1)
   InX3_c17 <= Out1_c10; //R(C1)
   InA4_c17 <= RW2; //R(W2)
   InA3_c17 <= IW2; //I(W2)
   //Configuration of
   InA2_c18 <= Out1_c14; //R(D1)
InA1_c18 <= Out1_c15; //I(D1)
   InA0_c18 <= zero;
   InX5_c18 <= zero;
   InX2_c18 <= RW3; //R(W3)
   InX1_c18 <= IW3; //I(W3)
   InX0_c18 <= zero;
   InX4_c18 <= Out1_c15; //I(D1)
   InX3_c18 <= Out1_c14; //R(D1)
   InA4_c18 <= RW3; //R(W3)
   InA3_c18 <= IW3; //I(W3)
   //Configuration c19
   InA2_c19 <= Out1_c4; //R(B2)
   InA1_c19 <= Out1_c5; //I(B2)
   InA0_c19 <= moinsun;
   InX5_c19 <= zero;
   InX2_c19 <= RW2; //R(W2)
   InX1_c19 <= IW2; //I(W2)
   //R(C2)*(-1)=I(W4*R(C2))
   InX0_c19 <= Out1_c8;
   InX4_c19 <= Out1_c5; //I(B2)
   InX3_c19 <= Out1_c4; //R(B2)
   InA4_c19 <= RW2; //R(W2)
   InA3_c19 <= IW2; //I(W2)
    //Configuration of
   InA2_c20 <= Out1_c12; //R(D2)
   InA1_c20 <= Out1_c13; //I(D2)
   InA0_c20 <= zero;
   InX5_c20 <= zero;
   InX2_c20 <= RW6; //R(W6)
   InX1_c20 <= IW6; //I(W6)
   InX0_c20 <= zero;
   InX4_c20 <= Out1_c13; //I(D2)
   InX3_c20 <= Out1_c12; //R(D2)
   InA4_c20 <= RW6; //R(W6)
   InA3_c20 <= IW6; //I(W6)
   //Configuration of c21
   InA2_c21 <= Out0_c6; //R(B3)
   InA1_c21 <= Out0_c7; //I(B3)
   InA0_c21 <= zero;
   InX5_c21 <= zero;
   InX2_c21 <= RW3; //R(W3)
   InX1_c21 <= IW3; //I(W3)
   InX0_c21 <= zero;
   InX4_c21 <= Out0_c7; //I(B3)
   InX3_c21 <= Out0_c6; //R(B3)
   InA4_c21 <= RW3; //R(W3)
   InA3_c21 <= IW3; //I(W3)
   //Configuration of c22
   InA2_c22 <= Out0_c10; //R(C3)
   InA1_c22 <= Out0_c11; //I(C3)
   InA0_c22 <= zero;
   InX5_c22 <= zero;
   InX2_c22 <= RW6; //R(W6)
   InX1_c22 <= IW6; //I(W6)
   InX0_c22 <= zero;
   InX4_c22 <= Out0_c11; //I(C3)
   InX3_c22 <= Out0_c10; //R(C3)
   InA4_c22 <= RW6; //R(W6)
   InA3_c22 <= IW6; //I(W6)
    //Configuration of c23
   InA2_c23 <= Out0_c14; //R(D3)
   InA1_c23 <= Out0_c15; //I(D3)
   InA0_c23 <= zero;
   InX5_c23 <= zero;
   InX2_c23 <= RW9; //R(W9)
   InX1_c23 <= IW9; //I(W9)
   InX0_c23 <= zero;
   InX4_c23 <= Out0_c15; //I(D3)
   InX3_c23 <= Out0_c14; //R(D3)
   InA4_c23 <= RW9; //R(W9)
   InA3_c23 <= IW9; //I(W9)
   //FFT outputs
   //connected to //registers
   LatchA0 <= Out0_c0; //R(A0)
   LatchA1 <= Out0_c1; //I(A0)
   LatchA2 <= Out0_c4; //R(B0)
   LatchA3 <= Out0_c5; //I(B0)
   LatchA4 <= Out0_c8; //R(C0)
```

ANNEXE 34 – Algorithme du processeur matriciel *6*x4 UPMs (16)

```
LatchA5 <= Out0_c9; //I(C0)
   LatchA6 <= Out0_c12; //R(D0)
   LatchA7 <= Out0_c13; //I(D0)
   LatchA8 <= Out1_c2; //R(A1)
   LatchA9 <= Out1_c3; //I(A1)
   LatchA10 <= Out0_c16; //R(W*B1)
   LatchA11 <= Out4_c16; //I(W*B1)
   LatchA12 <= Out0_c17; //R(W*C1)
   LatchA13 <= Out4_c17; //I(W*C1)
   LatchA14 <= Out0_c18; //R(W*D1)
   LatchA15 <= Out0_c18; //I(W*D1)
   LatchA16 <= Out1_c0; //R(A2)
   LatchA17 <= Out1_c1; //I(A2)
   LatchA18 <= Out0_c19; //R(W*B2)
   LatchA19 <= Out4_c19; //I(WB2)
   LatchA20 <= Out1_c9; //R(W*C2)

//I(C2)=R(W4*R(C2))
   LatchA21 <= Out2_c19;
   LatchA22 <= Out0_c20; //R(W*D2)
   LatchA23 <= Out4_c20; //I(W*D2)
   LatchA24 <= Out0_c2; //R(A3)
   LatchA25 <= Out0_c3; //I(A3)
   LatchA26 <= Out0_c21; //R(W*B3)
   LatchA27 <= Out4_c21; //I(W*B3)
   LatchA28 <= Out1_c22; //R(W*C3)
   LatchA29 <= Out4_c22; //I(W*C3)
   LatchA30 <= Out0_c23; //R(W*D3)
   LatchA31 <= Out4_c23; //I(W*D3)
   //connected to multiplexer
   OutMux0 <= Out0_c0; //R(A0)
   OutMux1 <= Out0_c1; //I(A0)
   OutMux2 <= Out0_c4; //R(B0)
   OutMux3 <= Out0_c5; //I(B0)
   OutMux4 <= Out0_c8; //R(C0)
   OutMux5 <= Out0_c9; //I(C0)
   OutMux6 <= Out0_c12; //R(D0)
   OutMux7 <= Out0_c13; //I(D0)
   OutMux8 <= Out1_c2; //R(A1)
   OutMux9 <= Out1_c3; //I(A1)
   OutMux10 <= Out0_c16; //R(W*B1)
   OutMux11 <= Out4_c16; //I(W*B1)
   OutMux12 <= Out0_c17; //R(W*C1)
   OutMux13 <= Out4_c17; //I(W*C1)
   OutMux14 <= Out0_c18; //R(W*D1)
   OutMux15 <= Out0_c18; //I(W*D1)
   OutMux16 <= Out1_c0; //R(A2)
   OutMux17 <= Out1_c1; //I(A2)
   OutMux18 <= Out0_c19; //R(W*B2)
   OutMux19 <= Out4_c19; //I(W*B2)
   OutMux20 <= Out1_c9; //R(W*C2)
   OutMux21 <= Out2_c19; //I(W*C2)
   OutMux22 <= Out0_c20; //R(W*D2)
   OutMux23 <= Out4_c20; //I(W*D2)
   OutMux24 <= Out0_c2; //R(A3)
   OutMux25 <= Out0_c3; //I(A3)
   OutMux26 <= Out0_c21; //R(W*B3)
   OutMux27 <= Out4_c21; //I(W*B3)
   OutMux28 <= Out1_c22; //R(W*C3)
   OutMux29 <= Out4_c22; //I(W*C3)
   OutMux30 <= Out0_c23; //R(W*D3)
   OutMux31 <= Out4_c23; //I(W*D3)
   Output0 <= MuxOutput;
   Output1 <= Out4_c0;
         end
//////////////////////
//Second array of four///
//Radix-4 Butterfly//////
//////////////////////
  else if(bit2==1'b1)
begin
Control bits
//Configuration
bit1 <= 1'b1;
//
bit3_c0 <= 1'b0;
bit3_c1 <= 1'b0;
bit3_c2 <= 1'b0;
bit3_c3 <= 1'b0;
bit3_c4 <= 1'b0;
bit3_c5 <= 1'b0;
bit3_c6 <= 1'b0;
bit3_c7 <= 1'b0;
bit3_c8 <= 1'b0;
bit3_c9 <= 1'b0;
//C0 add_sub bits
   ad0_c0 <= 1'b1;
   ad1_c0 <= 1'b1;
   ad2_c0 <= 1'b0;
   ad3_c0 <= 1'b1;
   ad4_c0 <= 1'b1;
   ad5_c0 <= 1'b1;
   ////C1 add_sub bit
   ad0_c1 <= 1'b1;
   ad1_c1 <= 1'b1;
   ad2_c1 <= 1'b0;
   ad3_c1 <= 1'b1;
   ad4_c1 <= 1'b1;
   ad5_c1 <= 1'b1;
   //C2 add_sub bit
   ad0_c2 <= 1'b0;
   ad1_c2 <= 1'b1;
   ad2_c2 <= 1'b0;
   ad3_c2 <= 1'b0;
   ad4_c2 <= 1'b1;
   ad5_c2 <= 1'b1;
   //C3 add_sub bit
   ad0_c3 <= 1'b0;
   ad1_c3 <= 1'b1;
   ad2_c3 <= 1'b0;
   ad3_c3 <= 1'b0;
   ad4_c3 <= 1'b1;
   ad5_c3 <= 1'b1;
//C4 add_sub bits
ad0_c4 <= 1'b1;
ad1_c4 <= 1'b1;
ad2_c4 <= 1'b0;
ad3_c4 <= 1'b1;
ad4_c4 <= 1'b1;
ad5_c4 <= 1'b1;
////C5 add_sub bit
ad0_c5 <= 1'b1;
ad1_c5 <= 1'b1;
ad2_c5 <= 1'b0;
ad3_c5 <= 1'b1;
ad4_c5 <= 1'b1;
ad5_c5 <= 1'b1;
//C6 add_sub bit
ad0_c6 <= 1'b0;
ad1_c6 <= 1'b1;
ad2_c6 <= 1'b0;
ad3_c6 <= 1'b0;
ad4_c6 <= 1'b1;
ad5_c6 <= 1'b1;
//C7 add_sub bit
ad0_c7 <= 1'b0;
ad1_c7 <= 1'b1;
ad2_c7 <= 1'b0;
ad3_c7 <= 1'b0;
ad4_c7 <= 1'b1;
ad5_c7 <= 1'b1;
//C8 add_sub bits
ad0_c8 <= 1'b1;
ad1_c8 <= 1'b1;
ad2_c8 <= 1'b0;
ad3_c8 <= 1'b1;
ad4_c8 <= 1'b1;
ad5_c8 <= 1'b1;
////C9 add_sub bit
ad0_c9 <= 1'b1;
ad1_c9 <= 1'b1;
ad2_c9 <= 1'b0;
ad3_c9 <= 1'b1;
ad4_c9 <= 1'b1;
ad5_c9 <= 1'b1;
//C10 add_sub bit
ad0_c10 <= 1'b0;
ad1_c10 <= 1'b1;
ad2_c10 <= 1'b0;
ad3_c10 <= 1'b0;
//C11 add_sub bit
ad0_c11 <= 1'b0;
ad1_c11 <= 1'b1;
ad2_c11 <= 1'b0;
ad3_c11 <= 1'b0;
 //C12 add_sub bits
ad0_c12 <= 1'b1;
ad1_c12 <= 1'b1;
ad2_c12 <= 1'b0;
ad3_c12 <= 1'b1;
////C13 add_sub bit
```

ANNEXE 35 – Algorithme du processeur matriciel *6*x4 UPMs (17)

```
ad0_c13  <= 1'b1;
   ad1_c13  <= 1'b1;
   ad2_c13  <= 1'b0;
   ad3_c13  <= 1'b1;
   //C14 add_sub bit
   ad0_c14  <= 1'b0;
   ad1_c14  <= 1'b1;
   ad2_c14  <= 1'b0;
   ad3_c14  <= 1'b0;
   //C15 add_sub bit
   ad0_c15  <= 1'b0;
   ad1_c15  <= 1'b1;
   ad2_c15  <= 1'b0;
   ad3_c15  <= 1'b0;
 //Configuration of
   // 2 UPMs c0 and c1
   InA2_c0 <= un;
   InA1_c0 <= un;
   InA0_c0 <= two;
   InX5_c0 <= LatchB0;
//R(A0)
   InX2_c0 <= LatchB4;
//R(C0)
   InX1_c0 <= Out4_c0;
//R(D0+B0)
   InX0_c0 <= Out4_c0;
//R(D0+B0)
   InX4_c0 <= LatchB2;
//R(B0)
   InX3_c0 <= LatchB6;
//R(D0)
   InA4_c0 <= un;
   InA3_c0 <= un;
   InD2_c0 <= zero;
   InX6_c0 <= zero;
   //
   InA2_c1 <= un;
   InA1_c1 <= un;
   InA0_c1 <= two;
   InX5_c1 <= LatchB1;
//I(A0)
   InX2_c1 <= LatchB5;
//I(C0)
   InX1_c1 <= Out4_c1;
//I(D0+B0)
   InX0_c1 <= Out4_c1;
//I(D0+B0)
   InX4_c1 <= LatchB3;
//I(B0)
   InX3_c1 <= LatchB7;
//I(D0)
   InA4_c1 <= un;
   InA3_c1 <= un;
   InD2_c1 <= zero;
   InX6_c1 <= zero;
   //Configuration of
   // 2 UPMs c18 and
   InA2_c2 <= un;
   InA1_c2 <= moinsun;
InA0_c2 <= moinstwo;
   InX5_c2 <= LatchB0;
//R(A0)
   InX2_c2 <= LatchB4;
//R(C0)
   InX1_c2 <= Out4_c3;
//I(B0-D0)
   InX0_c2 <= Out4_c3;
//I(B0-D0)
   InX4_c2 <= LatchB2;
//R(B0)
   InX3_c2 <= LatchB6;
//R(D0)
   InA4_c2 <= un;
   InA3_c2 <= un;
   InD2_c2 <= zero;
   InX6_c2 <= zero;
   //
   InA2_c3 <= un;
   InA1_c3 <= un;
   InA0_c3 <= two;
   InX5_c3 <= LatchB1;
//I(A0)
   InX2_c3 <= LatchB5;
//I(C0)
   InX1_c3 <= Out4_c2;
//R(B0-D0)
   InX0_c3 <= Out4_c2;
//R(B0-D0)
   InX4_c3 <= LatchB3;
//I(B0)
   InX3_c3 <= LatchB7;
//I(D0)
   InA4_c3 <= un;
   InA3_c3 <= un;
   InD2_c3 <= zero;
   InX6_c3 <= zero;
   //r=4 Butterfly 0
   //Out0_c0=R(F0)
   //Out0_c1=I(F0)
   //Out1_c2=R(F4)
   //Out1_c3=I(F4)
   //Out1_c0=R(F8)
   //Out1_c1=I(F8)
   //Out0_c2=R(F12)
   //Out0_c3=I(F12)
////////////////////
   //Butterfly 1//
   //Configuration of
   // 2 UPMs c23
   InA2_c4 <= un;
   InA1_c4 <= un;
   InA0_c4 <= two;
   InX5_c4 <= LatchB8;
//R(A1)
InX2_c4  <=  LatchB12;
//R(C1)
InX1_c4  <=  Out4_c4;
//R(D1+B1)
   InX0_c4 <= Out4_c4;
//R(D1+B1)
InX4_c4  <=  LatchB10;
//R(B1)
InX3_c4  <=  LatchB14;
//R(D1)
   InA4_c4 <= un;
   InA3_c4 <= un;
   InD2_c4 <= zero;
   InX6_c4 <= zero;
   //
   InA2_c5 <= un;
   InA1_c5 <= un;
   InA0_c5 <= two;
   InX5_c5 <= LatchB9;
//I(A1)
   InX2_c5          <=
LatchB13; //I(C1)
   InX1_c5 <= Out4_c5;
//I(D1+B1)
   InX0_c5 <= Out4_c5;
//I(D1+B1)
   InX4_c5          <=
LatchB11; //I(B1)
   InX3_c5          <=
LatchB15; //I(D1)
   InA4_c5 <= un;
   InA3_c5 <= un;
   InD2_c5 <= zero;
   InX6_c5 <= zero;
//Configuration of
   // 2 UPMs c25 and
InA2_c6 <= un;
InA1_c6 <= moinsun;
InA0_c6 <= moinstwo;
   InX5_c6 <= LatchB8;
//R(A1)
   InX2_c6          <=
LatchB12; //R(C1)
   InX1_c6 <= Out4_c7;
//I(B1-D1)
   InX0_c6 <= Out4_c7;
//I(B1-D1)
   InX4_c6          <=
LatchB10; //R(B1)
   InX3_c6          <=
LatchB14; //R(D1)
   InA4_c6 <= un;
   InA3_c6 <= un;
   InD2_c6 <= zero;
   InX6_c6 <= zero;
   //
   InA2_c7 <= un;
   InA1_c7 <= un;
   InA0_c7 <= two;
   InX5_c7 <= LatchB9;
//I(A1)
   InX2_c7          <=
LatchB13; //I(C1)
   InX1_c7 <= Out4_c6;
//R(B1-D1)
   InX0_c7 <= Out4_c6;
//R(B1-D1)
   InX4_c7          <=
LatchB11; //I(B1)
   InX3_c7          <=
LatchB15; //I(D1)
   InA4_c7 <= un;
   InA3_c7 <= un;
   InD2_c7 <= zero;
   InX6_c7 <= zero;
   //r=4 Butterfly 1
 //Out0_c4=R(F1)
//Out0_c5=I(F1)
//Out1_c6=R(F5)
//Out1_c7=I(F5)
   //Out1_c4=R(F9)
         //Out1_c5=I(F
9)
         //Out0_c6=R(
F13)
   //Out0_c7=I(F13)
////////////////////
   //Butterfly 2//
   //Configuration of
   // 2 UPMs c30 and
c31
   InA2_c8 <= un;
   InA1_c8 <= un;
   InA0_c8 <= two;
   InX5_c8          <=
LatchB16; //R(A2)
   InX2_c8          <=
LatchB20; //R(C2)
   InX1_c8 <= Out4_c8;
//R(D2+B2)
   InX0_c8 <= Out4_c8;
//R(D2+B2)
   InX4_c8          <=
LatchB18; //R(B2)
   InX3_c8          <=
LatchB22; //R(D2)
   InA4_c8 <= un;
   InA3_c8 <= un;
   InD2_c8 <= zero;
   InX6_c8 <= zero;
   //
   InA2_c9 <= un;
   InA1_c9 <= un;
   InA0_c9 <= two;
   InX5_c9          <=
LatchB17; //I(A2)
   InX2_c9          <=
LatchB21; //I(C2)
   InX1_c9  <=  Out4_c9
; //I(D2+B2)
   InX0_c9  <=  Out4_c9
; //I(D2+B2)
```

ANNEXE 36 – Algorithme du processeur matriciel *6*x4 UPMs (18)

```
 InX4_c9 <= LatchB19; //I(B2)
   InX3_c9 <= LatchB23; //I(D2)
   InA4_c9 <= un;
   InA3_c9 <= un;
   InD2_c9 <= zero;
   InX6_c9 <= zero;
//Configuration of
   // 2 UPMs c32 and
InA2_c10 <= un;
InA1_c10 <= moinsun;
InA0_c10 <= moinstwo;
   InX5_c10 <= LatchB16; //R(A2)
   InX2_c10 <= LatchB20; //R(C2)
   InX1_c10 <= Out4_c11; //I(B2-D2)
   InX0_c10 <= Out4_c11; //I(B2-D2)
   InX4_c10 <= LatchB18; //R(B2)
   InX3_c10 <= LatchB22; //R(D2)
   InA4_c10 <= un;
   InA3_c10 <= un;
   ///
   InA2_c11 <= un;
   InA1_c11 <= un;
   InA0_c11 <= two;
   InX5_c11 <= LatchB17; //I(A2)
   InX2_c11 <= LatchB21; //I(C2)
   InX1_c11 <= Out4_c10; //R(B2-D2)
   InX0_c11 <= Out4_c10; //R(B2-D2)
   InX4_c11 <= LatchB19; //I(B2)
   InX3_c11 <= LatchB23; //I(D2)
   InA4_c11 <= un;
   InA3_c11 <= un;
   //r=4 Butterfly 2 outputs
   //Out0_c8=R(F2)
   //Out0_c9=I(F2)
   //Out1_c10=R(F6)
//Out1_c11=I(F6)
//Out1_c8=R(F10)
//Out1_c9=I(F10)
//Out0_c10=R(F14)
//Out0_c11=I(F14)
 ////////////////////
//Butterfly 3//
//Configuration of
// 2 UPMs c37 and c38
   InA2_c12 <= un;
   InA1_c12 <= un;
   InA0_c12 <= two;
   InX5_c12 <= LatchB24; //R(A3)
   InX2_c12 <= LatchB28; //R(C3)
InX1_c12 <= Out4_c12; //R(D3+B3)
InX0_c12 <= Out4_c12; //R(D3+B3)
   InX4_c12 <= LatchB26; //R(B3)
   InX3_c12 <= LatchB30; //R(D3)
   InA4_c12 <= un;
   InA3_c12 <= un;
   //
   InA2_c13 <= un;
   InA1_c13 <= un;
   InA0_c13 <= two;
   InX5_c13 <= LatchB25; //I(A3)
   InX2_c13 <= LatchB29; //I(C3)
   InX1_c13 <= Out4_c13 ;//I(D3+B3)
   InX0_c13 <= Out4_c13 ;//I(D3+B3)
   InX4_c13 <= LatchB27; //I(B3)
   InX3_c13 <= LatchB31; //I(D3)
   InA4_c13 <= un;
   InA3_c13 <= un;
//Configuration of
// 2 UPMs c18 and c19
   InA2_c14 <= un;
InA1_c14 <= moinsun;
InA0_c14 <= moinstwo;
   InX5_c14 <= LatchB24; //R(A3)
   InX2_c14 <= LatchB28; //R(C3)
   InX1_c14 <= Out4_c15; //I(B3-D3)
   InX0_c14 <= Out4_c15; //I(B3-D3)
   InX4_c14 <= LatchB26; //R(B3)
   InX3_c14 <= LatchB30; //R(D3)
   InA4_c14 <= un;
   InA3_c14 <= un;
   //
   InA2_c15 <= un;
   InA1_c15 <= un;
   InA0_c15 <= two;
   InX5_c15 <= LatchB25; //I(A2)
   InX2_c15 <= LatchB29; //I(C3)
   InX1_c15 <= Out4_c14; //R(B3-D3)
   InX0_c15 <= Out4_c14; //R(B3-D3)
   InX4_c15 <= LatchB27; //I(B3)
   InX3_c15 <= LatchB31; //I(D3)
   InA4_c15 <= un;
   InA3_c15 <= un;
////////////////////////////////////
////
////////////////////////////////////
////
    //FFT outputs
    //connected to multiplexer
   OutMux0 <= Out0_c0;  //R(F0)
   OutMux1 <= Out0_c1;  //I(F0)
   OutMux2 <= Out0_c4;  //R(F1)
   OutMux3 <= Out0_c5;  //I(F1)
   OutMux4 <= Out0_c8;  //R(F2)
   OutMux5 <= Out0_c9;  //I(F2)
   OutMux6 <= Out0_c12;  //R(F3)
   OutMux7 <= Out0_c13;  //I(F3)
   OutMux8 <= Out1_c2;  //R(F4)
   OutMux9 <= Out1_c3;  //I(F4)
   OutMux10 <= Out1_c6;  //R(F5)
   OutMux11 <= Out1_c7;  //I(F5)
   OutMux12 <= Out1_c10; //R(F6)
   OutMux13 <= Out1_c11; //I(F6)
   OutMux14 <= Out1_c14; //R(F7)
   OutMux15 <= Out1_c15; //I(F7)
   OutMux16 <= Out1_c0;  //R(F8)
   OutMux17 <= Out1_c1;  //I(F8)
   OutMux18 <= Out1_c4;  //R(F9)
   OutMux19 <= Out1_c5;  //I(F9)
   OutMux20 <= Out1_c8;  //R(F10)
   OutMux21 <= Out1_c9;  //I(F10)
   OutMux22 <= Out1_c12; //R(F11)
   OutMux23 <= Out1_c13; //I(F11)
   OutMux24 <= Out0_c2;  //R(F12)
   OutMux25 <= Out0_c3;  //I(F12)
   OutMux26 <= Out0_c6;  //R(F13)
   OutMux27 <= Out0_c7;  //I(F13)
   OutMux28 <= Out0_c10; //R(F14)
   OutMux29 <= Out0_c11; //I(F14)
   OutMux30 <= Out0_c14; //R(F15)
   OutMux31 <= Out0_c15; //I(F15)
Output0 <= MuxOutput;
Output1 <= zero;
end
end

end

endmodule
```

ANNEXE 37 – Algorithme de la fonction UPMR.v

```
////////////////////////////////////UPM.v//////////////////////////////////////////////////
//Universal Processing Module instantiation
module UPMR (InA2, InA1,InA0, InX5, InX2, InX1, InX0, InX4, InX3, InX6, InA4, InA3, InD2,
                                FP_clock, bit1, bit2, bit3, ad0, ad1, ad2, ad3, ad4, ad5,
        Out0, Out1, Out4, Out3, Out2, Out5, Out6, result2);  //Fuction instantiation
//Inputs data
input       [31:0]  InA2, InA1,InA0, InX5, InX2, InX1, InX0, InX4, InX3, InX6, InA4, InA3, InD2;
input               FP_clock, bit1, bit2, bit3, ad0, ad1, ad2, ad3, ad4, ad5;  //Control bits
output      [31:0]      Out0, Out1, Out4, Out3, Out2, Out5, Out6, result2;       //Output Data
wire    [31:0]  Out0, Out1, Out4, Out3, Out2, Out5, Out6;
wire    [31:0]  result0, result1, result2;     //Internal connections
wire    [31:0]  result3, result4, result5;     //Internal connections
wire    [31:0]  reg0, reg1, reg2, reg3, reg4;  //Internal connections
wire    [31:0]  re0, re1, re2;                                  //Internal connections
///Function calls
//Recursive control call
ControleRecurs  cr (InX4, InX3, Out3, InX6, bit1, bit2, bit3, reg3, reg4, Out5, Out6);
Registre32b     r1 (bit2, InA2, bit2, result0);                                 //Latch function
Registre32b     r2 (bit2, result0, bit2, result1);       //Latch function
Registre32b     r3 (bit2, result1, bit2, result2);         //Latch function
Multiplexeur32b m1 (InA2, result0, ~bit1, result3); //Multiplexer Function
Multiplexeur32b m2 (InA1, result1, ~bit1, result4); //Multiplexer Function
Multiplexeur32b m3 (InA0, result2, ~bit1, result5); //Multiplexer Function
FP_MULT0        mu1             (FP_clock, result3, InX2, reg0);   ///32-bits Floating point multiplier
FP_ADD_SUB      su1  (ad0, FP_clock, InX5, reg0, reg1); ///32-bits Floating point adder
FP_MULT0        mu2             (FP_clock, result4, InX1, reg2);  ///32-bits Floating point multiplier
FP_ADD_SUB      su2  (ad1, FP_clock, reg1, reg2, Out0); ///32-bits Floating point adder
FP_MULT0        mu3             (FP_clock, result5, InX0, Out2);  ///32-bits Floating point multiplier
FP_ADD_SUB      su3  (ad2, FP_clock, Out0, Out2, Out1); ///32-bits Floating point adder
FP_MULT0        mu4             (FP_clock, InA4, reg3, re0);  ///32-bits Floating point multiplier
FP_MULT         mu5             (FP_clock, InA3, reg4, re1);  ///32-bits Floating point multiplier
FP_ADD_SUB      su4  (ad3, FP_clock, re0, re1, re2); ///32-bits Floating point adder
FP_ADD_SUB      su5  (ad4, FP_clock, re2, InD2, Out4); ///32-bits Floating point adder
FP_ADD_SUB      su6  (ad5, FP_clock, Out1, Out4, Out3); ///32-bits Floating point adder

Endmodule
```

ANNEXE 38 – Algorithme de la fonction UPM.v

```
////////////////////////////////////UPM.v//////////////////////////////////////////////////

//Universal Processing Module instantiation
module UPM (InA2, InA1,InA0, InX5, InX2, InX1, InX0, InX4, InX3, InA4, InA3,
                                FP_clock, ad0, ad1, ad2, ad3, Out0, Out1, Out4, Out2);  //Fuction instantiation
//Input datas
input       [31:0]  InA2, InA1,InA0, InX5, InX2, InX1, InX0, InX4, InX3, InA4, InA3;
input               FP_clock, ad0, ad1, ad2, ad3;  //Control bits
output      [31:0]      Out0, Out1, Out4, Out2;        //Output Data
wire    [31:0]  Out0, Out1, Out4, Out2;
wire    [31:0]  reg0, reg1, reg2;  //Internal connections
wire    [31:0]  re0, re1, re2;                                  //Internal connections
///Function calls
FP_MULT0        mu1             (FP_clock, InA2, InX2, reg0);   ///32-bits Floating point multiplier
FP_ADD_SUB      su1  (ad0, FP_clock, InX5, reg0, reg1); ///32-bits Floating point adder
FP_MULT0        mu2             (FP_clock, InA1, InX1, reg2);  ///32-bits Floating point multiplier
FP_ADD_SUB      su2  (ad1, FP_clock, reg1, reg2, Out0); ///32-bits Floating point adder
FP_MULT         mu3             (FP_clock, InA0, InX0, Out2);  ///32-bits Floating point multiplier
FP_ADD_SUB      su3  (ad2, FP_clock, Out0, Out2, Out1); ///32-bits Floating point adder
FP_MULT         mu4             (FP_clock, InA4, InX4, re0);  ///32-bits Floating point multiplier
FP_MULT         mu5             (FP_clock, InA3, InX3, re1);  ///32-bits Floating point multiplier
FP_ADD_SUB      su4  (ad3, FP_clock, re0, re1, Out4); ///32-bits Floating point adder

endmodule
```

ANNEXE 39 – Algorithme Matlab de reconfiguration des connections (1)

```
function [InA2_c0, InA2_c1, InA2_c2, InA2_c3, InA1_c0, InA1_c1, InA1_c2,  ...
InA0_c0, InA0_c1, InA0_c2, InA0_c3, InX5_c0, InX5_c1, InX5_c2, InX5_c3, ...
InX2_c0, InX2_c1, InX2_c2, InX2_c3, InX1_c0, InX1_c1, InX1_c2, InX1_c3, ...
InX0_c0, InX0_c1, InX0_c2, InX0_c3, InX4_c0, InX4_c1, InX4_c2, InX4_c3, ...
InX3_c0, InX3_c1, InX3_c2, InX3_c3, InA4_c0, InA4_c1, InA4_c2, InA4_c3, ...
InA3_c0, InA3_c1, InA3_c2, InA3_c3, InD2_c0, InD2_c1, InD2_c2, InD2_c3, ...
InX6_c0, InX6_c1, InX6_c2, InX6_c3, dataI2, InA1_c3,  ...
bit3_c0, ad1_c0,ad3_c0, ad5_c0, Output0, Output1, ...
bit3_c1, bit3_c2, bit3_c3, bit1, ad0_c0, ad0_c1, ad0_c2, ad0_c3, ...
ad1_c1, ad1_c2, ad1_c3, ad2_c0, ad2_c1, ad2_c2, ad2_c3, ...
ad3_c1, ad3_c2, ad3_c3, ad4_c0, ad4_c1, ad4_c2, ad4_c3, ...
ad5_c1, ad5_c2, ad5_c3, InMux0, InMux1, InMux2, InMux3, InMux4, InMux5,
InMux6, ...
InMux7] = ConnectionsReconfig( clk, Samples, Samples1, Opcode, ...
Out0_c0, Out0_c1, Out0_c2, Out0_c3, Out1_c0, Out1_c1, Out1_c2, Out1_c3, ...
Out2_c0, Out2_c1, Out2_c2, Out2_c3,  Out3_c0, Out3_c1, Out3_c2, Out3_c3, ...
Out4_c0, Out4_c1, Out4_c2, Out4_c3, Out5_c0, Out5_c1, Out5_c2, Out5_c3, ...
Out6_c0, Out6_c1, Out6_c2, Out6_c3, result2_c0, result2_c1,...
data1, data2, data3, data4, data5, data6, data7, data8, ...
data9, data10, data11, data12, data13, data14, data15, ...
datax1, MuxOutput, data16, data17)

InA2_c0=0;
InA2_c1=0;
InA2_c2=0;
InA2_c3=0;
InA1_c0=0;
InA1_c1=0;
InA1_c2=0;
InA1_c3=0;
InA0_c0=0;
InA0_c1=0;
InA0_c2=0;
InA0_c3=0;
InX5_c0=0;
InX5_c1=0;
InX5_c2=0;
InX5_c3=0;
InX2_c0=0;
InX2_c1=0;
InX2_c2=0;
InX2_c3=0;
InX1_c0=0;
InX1_c1=0;
InX1_c2=0;
InX1_c3=0;
InX0_c0=0;
InX0_c1=0;
InX0_c2=0;
InX0_c3=0;
InX4_c0=0;
InX4_c1=0;
InX4_c2=0;
InX4_c3=0;
InX3_c0=0;
InX3_c1=0;
InX3_c2=0;
InX3_c3=0;
InA4_c0=0;
InA4_c1=0;
InA4_c2=0;
InA4_c3=0;
InA3_c0=0;
InA3_c1=0;
InA3_c2=0;
InA3_c3=0;
InD2_c0=0;
InD2_c1=0;
InD2_c2=0;
InD2_c3=0;
InX6_c0=0;
InX6_c1=0;
InX6_c2=0;
InX6_c3=0;
dataI2=0;
bit3_c0=0;
ad1_c0=0;
ad3_c0=0;
ad5_c0=0;
Output0=0;
Output1=0;
bit3_c1=0;
bit3_c2=0;
bit3_c3=0;
bit1=0;
ad0_c0=0;
ad0_c1=0;
ad0_c2=0;
ad0_c3=0;
ad1_c1=0;
ad1_c2=0;
ad1_c3=0;
ad2_c0=0;
ad2_c1=0;
ad2_c2=0;
ad2_c3=0;
ad3_c1=0;
ad3_c2=0;
ad3_c3=0;
ad4_c0=0;
ad4_c1=0;
ad4_c2=0;
ad4_c3=0;
ad5_c1=0;
ad5_c2=0;
ad5_c3=0;
InMux0=0;
InMux1=0;
InMux2=0;
InMux3=0;
InMux4=0;
InMux5=0;
InMux6=0;
InMux7=0;


 if
(Opcode==0)

if(clk==1)
    %NON-
ADAPTIVE FIR-
IIR
%CORRELATOR
N=20 processor
input
InA2_c0 =
Samples;
%Control bits
bit1    = 0;
bit3_c0 = 1;
bit3_c1 = 1;
bit3_c2 = 1;
bit3_c3 = 1;
```

ANNEXE 40 – Algorithme Matlab de reconfiguration des connections (2)

```
ad0_c0  = 1;
ad1_c0  = 1;
ad2_c0  = 1;
ad3_c0  = 1;
ad4_c0  = 1;
ad5_c0  = 0;
ad0_c1  = 1;
ad1_c1  = 1;
ad2_c1  = 1;
ad3_c1  = 1;
ad4_c1  = 1;
ad5_c1  = 0;
ad0_c2  = 1;
ad1_c2  = 1;
ad2_c2  = 1;
ad3_c2  = 1;
ad4_c2  = 1;
ad5_c2  = 0;
ad0_c3  = 1;
ad1_c3  = 1;
ad2_c3  = 1;
ad3_c3  = 1;
ad4_c3  = 1;
ad5_c3  = 0;
%UPM c0
parameters
InA1_c0 = 0;
InA0_c0 = 0;
InX5_c0 = 0;
InX2_c0 = 1;
InX1_c0 = 1;
InX0_c0 = 1;
InX4_c0 = 0;
InX3_c0 = 0;
InA4_c0 = 1;
InA3_c0 = 1;
InD2_c0 = 0;

%Interconnecti
on with
neighbourhood
UPM
InX6_c0 =
Out3_c3;
    InX5_c1 =
Out1_c0;
    InD2_c1 =
Out4_c0;
    InX6_c1 =
Out6_c0;

%Initializatio
n of UPM c1
parameters
    InA2_c1 =
result2_c0;
InA1_c1 = 0;
InA0_c1 = 0;
InX2_c1 = 1;
InX1_c1 = 1;
InX0_c1 = 1;
InX4_c1 = 0;
InX3_c1 = 0;
InA4_c1 = 1;
InA3_c1 = 1;

%Interconnecti
on with
neighbourhood
UPM
    InX5_c2 =
Out1_c1;
    InD2_c2 =
Out4_c1;
    InX6_c2 =
Out6_c1;
%UPM c2
parameters
InA2_c2 = 0;
InA1_c2 = 0;
InA0_c2 = 0;
InX2_c2 = 0;
InX1_c2 = 0;
InX0_c2 = 0;
InX4_c2 = 0;
InX3_c2 = 0;
InA4_c2 = 1;
InA3_c2 = 1;

%Interconnecti
on with
neighbourhood
UPM
    InX5_c3 =
Out1_c2;
    InD2_c3 =
Out4_c2;
    InX6_c3 =
Out6_c2;
    %UPM c3
parameters
    InA2_c3 =
result2_c1;
InA1_c3 = 0;
InA0_c3 = 0;
InX2_c3 = 1;
InX1_c3 = 1;
InX0_c3 = 1;
InX4_c3 = 0;
InX3_c3 = 0;
InA4_c3 = 1;
InA3_c3 = 1;
 %FIR IIR
Array outputs
Output0 =
Out1_c3;
%IIR output
Output1 =
Out3_c3;

elseif(clk==0)
     end
%%%%%%%%%%%%%%
%
%%%ADAPTIVE
FIR%%%%
%%& CORRELATOR
MODE N=9%%
%%%%%%%%%%%%%%
%
elseif
(Opcode==1)
  %ADAPTIVE FIR
  %CORRELATOR
N=9
  % processor
input
   InA2_c0 =
Samples;
  %Control bits
  bit1    = 0;
  bit3_c0 = 1;
  bit3_c1 = 1;
  bit3_c2 = 1;
  bit3_c3 = 1;
  ad0_c0  = 1;
  ad1_c0  = 1;
  ad2_c0  = 1;
  ad3_c0  = 1;
  ad4_c0  = 1;
  ad5_c0  = 0;
  ad0_c1  = 1;
  ad1_c1  = 1;
  ad2_c1  = 1;
  ad3_c1  = 1;
  ad4_c1  = 1;
  ad5_c1  = 0;
  ad0_c2  = 1;
  ad1_c2  = 1;
  ad2_c2  = 1;
  ad3_c2  = 1;
  ad4_c2  = 1;
  ad5_c2  = 0;
  ad0_c3  = 1;
  ad1_c3  = 1;
  ad2_c3  = 1;
  ad3_c3  = 1;
  ad4_c3  = 1;
  ad5_c3  = 0;

%Initializatio
n of
  %UPM c0
parameters
  InA1_c0 = 0;
  InA0_c0 = 0;
  InX5_c0 = 0;
  InX2_c0 =
data1;
  InX1_c0 =
data2;
  InX0_c0 =
data3;
  InX4_c0 = 0;
  InX3_c0 = 0;
  InA4_c0 = 0;
  InA3_c0 = 0;
  InD2_c0 = 0;

%Interconnecti
on
  %with
neighborhood
UPM
  InX6_c0 =
Out3_c3;
  InX5_c1 =
Out1_c0;
  InD2_c1 =
Out4_c0;
  InX6_c1 =
Out6_c0
```

ANNEXE 41 – Algorithme Matlab de reconfiguration des connections (3)

```
%Initialization
    %of UPM c1
  parameters
    InA2_c1 =
  result2_c0;
    InA1_c1 = 0;
    InA0_c1 = 0;
    InX2_c1 =
  data6;
    InX1_c1 =
  data7;
    InX0_c1 =
  data8;
    InX4_c1 = 0;
    InX3_c1 = 0;
    InA4_c1 = 0;
    InA3_c1 = 0;

  %Interconnecti
  on
    %with
  neighborhood
  UPM
    InX5_c2 =
  Out1_c1;
    InD2_c2 =
  Out4_c1;
    InX6_c2 =
  Out6_c1;

  %Initializatio
  n
    %of UPM c2
  parameters
    InA2_c2 =
  result2_c1;
    InA1_c2 = 0;
    InA0_c2 = 0;
    InX2_c2 =
  data9;
    InX1_c2 =
  data10;
    InX0_c2 =
  data11;
    InX4_c2 = 0;
    InX3_c2 = 0;
    InA4_c2 = 0;
    InA3_c2 = 0;

%Interconnecti
on
  %with
neighborhood
UPM
  InX5_c3 =
Out1_c2;
  InD2_c3 =
Out4_c2;
  InX6_c3 =
Out6_c2;

%Initializatio
n of
  %UPM c3
parameters
  InA2_c3 =
result2_c2;
  InA1_c3 = 0;
  InA0_c3 = 0;
  InX2_c3 =
data12;
  InX1_c3 =
data13;
  InX0_c3 =
data14;
  InX4_c3 = 0;
  InX3_c3 = 0;
  InA4_c3 = 0;
  InA3_c3 = 0;
   %FIR IIR
Array
processor
outputs
  Output0 =
Out1_c3;
  %IIR output
delayed by one
   Output1 =
 Out3_c3;

%%%%%%%%%%%%%%%
%%%ALL-POLE
 LATTICE MODE%%
%%%%%%%%%%%%%%%
 elseif
 (Opcode==2)
     if(clk==1)
     %ALL-POLE
 LATTICE input
    InX2_c0 =
 Samples;
    %Control
 bits
    bit1 = 1;
    bit3_c0 = 1;
    bit3_c1 = 1;
    bit3_c2 = 1;
    bit3_c3 = 1;
    %Add%Sub
 bits
    ad0_c0  = 1;
    ad1_c0  = 0;
    ad2_c0  = 1;
    ad3_c0  = 1;
    ad4_c0  = 1;
    ad5_c0  = 0;
    ad0_c1  = 1;
    ad1_c1  = 0;
    ad2_c1  = 1;
    ad3_c1  = 1;
    ad4_c1  = 1;
    ad5_c1  = 0;
    ad0_c2  = 1;
    ad1_c2  = 0;
    ad2_c2  = 1;
    ad3_c2  = 1;
    ad4_c2  = 1;
    ad5_c2  = 0;
    ad0_c3  = 1;
    ad1_c3  = 0;
    ad2_c3  = 1;
    ad3_c3  = 1;
    ad4_c3  = 1;
    ad5_c3  = 0;
 %%UPM c0
parameters
    InA2_c0 = 1;
    InA1_c0 = 1;
    InA0_c0 = 0;
    InX5_c0 = 0;
    InX0_c0 = 0;
    InA4_c0 = 1;
    InA3_c0 = 1;
    InD2_c0 = 0;

%Interconnecti
on
    %with
neighbourhood
    InX1_c0 =
Out5_c0;
    InX4_c0 =
Out5_c0;
    InX3_c0 =
Out0_c0;
    InX6_c0 =
Out4_c1;
    %Next UPM
input
    InX2_c1 =
Out0_c0;
    %UPM c1
parameters
    InA2_c1 = 1;
    InA1_c1 = 1;
    InA0_c1 = 0;
    InX5_c1 = 0;
    InX0_c1 = 0;
    InA4_c1 = 1;
    InA3_c1 = 1;
    InD2_c1 = 0;
```

ANNEXE 42 – Algorithme Matlab de reconfiguration des connections (4)

```
%Interconnexion
    %with
 neighbourhood
 inputs
    InX1_c1 =
 Out5_c1;
    InX4_c1 =
 Out5_c1;
    InX3_c1 =
 Out0_c1;
    InX6_c1 =
 Out4_c2;
%Next UPM input
    InX2_c2 =
 Out0_c1;
    %UPM c2
 parameters
    InA2_c2 = 1;
    InA1_c2 = 1;
    InA0_c2 = 0;
    InX5_c2 = 0;
    InX0_c2 = 0;
    InA4_c2 = 1;
    InA3_c2 = 1;
    InD2_c2 = 0;
%Interconnection
    %with
 neighbourhood
    InX1_c2 =
 Out5_c2;
    InX4_c2 =
 Out5_c2;
    InX3_c2 =
 Out0_c2;
    InX6_c2 =
 Out4_c3;
    %Next UPM
 input
    InX2_c3 =
 Out0_c2;

 %Initializatio
 n of
    %UPM c2
 parameters
    InA2_c3 = 1;
    InA1_c3 = 1;
    InA0_c3 = 0;
    InX5_c3 = 0;
    InX0_c3 = 0;
    InA4_c3 = 1;
    InA3_c3 = 1;
    InD2_c3 = 0;

 %Interconnecti
 on with
 neighbourhood
 inputs
    InX1_c3 =
 Out5_c3;
    InX4_c3 =
 Out5_c3;
    InX3_c3 =
 Out0_c3;
    InX6_c3 =
 Out0_c3;
%ALL-POLE
 LATTICE
output
    Output0 =
 Out0_c3;
    Output1 =
 Out4_c3;

 elseif(clk==0)
     end
%%%%%%%%%%%%%%%
%%%ALL-0 LATTICE
 MODE%%
%%%%%%%%%%%%%%%
 elseif
 (Opcode==3)
     %Processor
 inputs
    InX2_c0 =
 Samples;
    InX1_c0 =
 datax1;
    InX4_c0 =
 datax1;
    InX3_c0 =
 Samples;
    %Control
 bits
    bit1 = 1;
    bit3_c0 = 1;
    bit3_c1 = 1;
    bit3_c2 = 1;
    bit3_c3 = 1;
%Add%Sub bits
    ad0_c0  = 1;
    ad1_c0  = 1;
    ad2_c0  = 1;
    ad3_c0  = 1;
    ad4_c0  = 1;
    ad5_c0  = 1;
    ad0_c1  = 1;
    ad1_c1  = 1;
    ad2_c1  = 1;
    ad3_c1  = 1;
    ad4_c1  = 1;
    ad5_c1  = 1;
    ad0_c2  = 1;
    ad1_c2  = 1;
    ad2_c2  = 1;
    ad3_c2  = 1;
    ad4_c2  = 1;
    ad5_c2  = 1;
    ad0_c3  = 1;
    ad1_c3  = 1;
    ad2_c3  = 1;
    ad3_c3  = 1;
    ad4_c3  = 1;
    ad5_c3  = 1;
 %UPM c0
 parameters
    InA2_c0 =
data1;
    InA1_c0 =
data2;
    InA0_c0 = 0;
    InX5_c0 = 0;
    InX0_c0 = 0;
    InA4_c0 =
data3;
    InA3_c0 =
data4;
    InD2_c0 = 0;

%Interconnecti
on with
 neighbourhood
 inputs
    InX6_c0 =
Out4_c0;
    %Next UPM
inputs
    InX2_c1 =
Out0_c0;
    InX3_c1 =
Out0_c0;
    InX1_c1 =
Out5_c0;
    InX4_c1 =
Out5_c0;
    %UPM c1
parameters
    InA2_c1 =
data5;
    InA1_c1 =
data6;
    InA0_c1 = 0;
    InX5_c1 = 0;
    InX0_c1 = 0;
    InA4_c1 =
data7;
    InA3_c1 =
data8;
    InD2_c1 = 0;

%Interconnecti
on with
neighbourhood
inputs
    InX6_c1 =
Out4_c1;
     %Next UPM
inputs
    InX2_c2 =
Out0_c1;
    InX3_c2 =
Out0_c1;
    InX1_c2 =
Out5_c1;
    InX4_c2 =
Out5_c1;
```

ANNEXE 43 – Algorithme Matlab de reconfiguration des connections (5)

```
%Initializatio
n of UPM c2
parameters
   InA2_c2 =
data9;
   InA1_c2 =
data10;
   InA0_c2 = 0;
   InX5_c2 = 0;
   InX0_c2 = 0;
   InA4_c2 =
data11;
   InA3_c2 =
data12;
   InD2_c2 = 0;

%Interconnecti
on with
neighbourhood
inputs
   InX6_c2 =
Out4_c2;
   %Next UPM
inputs
   InX2_c3 =
Out0_c2;
   InX3_c3 =
Out0_c2;
   InX1_c3 =
Out5_c2;
   InX4_c3 =
Out5_c2;
   %UPM c3
parameters
   InA2_c3 =
data13;
   InA1_c3 =
data14;
   InA0_c3 = 0;
   InX5_c3 = 0;
   InX0_c3 = 0;
   InA4_c3 =
data15;
   InA3_c3 =
data16;
   InD2_c3 = 0;

%Interconnecti
on with
neighbourhood
inputs
   InX6_c3 =
Out4_c3;
    %ALL-0
LATTICE
outputs
   Output0 =
Out0_c3;
   Output1 =
Out4_c3;


%%%%%%%%%%%%%
%%%Radix-4
 Butterfly%%
%%%%%%%%%%%%%
 elseif
 (Opcode==4)
   %Butterfly
 r=4 mode
%Control bits%
   bit1 = 1;
   bit3_c0 = 0;
   bit3_c1 = 0;
   bit3_c2 = 0;
   bit3_c3 = 0;
 %C0 add_sub
 bits
   ad0_c0  = 1;
   ad1_c0  = 1;
   ad2_c0  = 0;
   ad3_c0  = 1;
   ad4_c0  = 1;
   ad5_c0  = 1;
   %%C1 add_sub
 bit
   ad0_c1  = 1;
   ad1_c1  = 1;
   ad2_c1  = 0;
   ad3_c1  = 1;
   ad4_c1  = 1;
   ad5_c1  = 1;
   %C2 add_sub
 bit
   ad0_c2  = 0;
   ad1_c2  = 1;
   ad2_c2  = 0;
   ad3_c2  = 0;
   ad4_c2  = 1;
   ad5_c2  = 1;
 %C3 add_sub
 bits
   ad0_c3  = 0;
   ad1_c3  = 1;
   ad2_c3  = 0;
   ad3_c3  = 0;
   ad4_c3  = 1;
   ad5_c3  = 1;
%Configuration
 of  2 UPMs c0
 and c1
   InA2_c0 = 1;
   InA1_c0 = 1;
   InA0_c0 = 2;
   InX5_c0 =
 data8;  %R(f0)
   InX2_c0 =
 data4;  %R(f2)
   InX1_c0 =
 Out4_c0;
 %R(s1)
   InX0_c0 =
 Out4_c0;
 %R(s1)
   InX4_c0 =
 data6;  %R(f1)
   InX3_c0 =
 data2;  %R(f3)
   InA4_c0 = 1;
   InA3_c0 = 1;
   InD2_c0 = 0;
   InX6_c0 = 0;
   %
   InA2_c1 = 1;
   InA1_c1 = 1;
   InA0_c1 = 2;
   InX5_c1 =
 data7;  %I(f0)
   InX2_c1 =
 data3;  %I(f2)
   InX1_c1 =
 Out4_c1;
 %I(s1)
   InX0_c1 =
 Out4_c1;
 %I(s1)
   InX4_c1 =
 data5;  %I(f1)
   InX3_c1 =
 data1;  %I(f3)
   InA4_c1 = 1;
   InA3_c1 = 1;
   InD2_c1 = 0;
   InX6_c1 = 0;

%Configuration
of
   % 2 UPMs c2
and c3
InA2_c2 = 1;
InA1_c2 = -1;
InA0_c2 = -2;
   InX5_c2 =
data8;  %R(f0)
   InX2_c2 =
data4;  %R(f2)
   InX1_c2 =
Out4_c3;
%I(s3)
   InX0_c2 =
Out4_c3;
%I(s3)
   InX4_c2 =
data6;  %R(f1)
   InX3_c2 =
data2;  %R(f3)
   InA4_c2 = 1;
   InA3_c2 = 1;
   InD2_c2 = 0;
   InX6_c2 = 0;
   %
   InA2_c3 = 1;
   InA1_c3 = 1;
   InA0_c3 = 2;
   InX5_c3 =
data7; %I(f0)
   InX2_c3 =
data3; %I(f2)
   InX1_c3 =
Out4_c2;%R(s3)
   InX0_c3 =
Out4_c2;%R(s3)
   InX4_c3 =
data5; %I(f1)
   InX3_c3 =
data1; %I(f3)
   InA4_c3 = 1;
   InA3_c3 = 1;
   InD2_c3 = 0;
   InX6_c3 = 0;
```

ANNEXE 44 – Algorithme Matlab de reconfiguration (6)

```
 %r=4 Butterfly
outputs
   InMux0 =
Out0_c0;
%R(F0)
   InMux1 =
Out0_c1;
%I(F0)
   InMux2 =
Out0_c2;
%R(F3)
   InMux3 =
Out0_c3;
%I(F3)
   InMux4 =
Out1_c0;
%R(F2)
   InMux5 =
Out1_c1;
%I(F2)
   InMux6 =
Out1_c2;
%R(F1)
   InMux7 =
Out1_c3;
%I(F1)
   Output0 =
MuxOutput;
   Output1 = 0;
%%%%%%%%%%%%%
%%%DIVISION%%%%
%%%%%%%%%%%%%
 elseif
 (Opcode==5)
   %Control
 bits
   bit1 = 1;
   bit3_c0 = 1;
   bit3_c1 = 1;
   bit3_c2 = 1;
   bit3_c3 = 1;
     %Add%Sub
      bits
   ad0_c0  = 1;
   ad1_c0  = 0;
   ad2_c0  = 1;
   ad3_c0  = 1;
   ad4_c0  = 1;
   ad5_c0  = 0;
   ad0_c1  = 1;
   ad1_c1  = 0;
   ad2_c1  = 1;
   ad3_c1  = 1;
   ad4_c1  = 1;
   ad5_c1  = 0;
   ad0_c2  = 1;
   ad1_c2  = 0;
   ad2_c2  = 1;
   ad3_c2  = 1;
   ad4_c2  = 1;
   ad5_c2  = 0;
   ad0_c3  = 1;
   ad1_c3  = 0;
   ad2_c3  = 1;
   ad3_c3  = 1;
   ad4_c3  = 1;
   ad5_c3  = 0;
%UPM c0
 parameters
   InA2_c0 =
data2;
   InA1_c0 =
datax1;
   InA0_c0 =
data2;
   InX5_c0 = 0;
   InX2_c0 = 2;
   InX1_c0 =
Out2_c0;
   InX0_c0 =
data2;
   InA4_c0 = 0;
   InA3_c0 = 0;
   InD2_c0 = 0;
   InX4_c0 = 0;
   InX3_c0 = 0;
   InX6_c0 = 0;
 %UPM c1
parameters
   InA2_c1 =
Out0_c0;
   InA1_c1 =
datax1;
   InA0_c1 =
Out0_c0;
   InX5_c1 = 0;
   InX2_c1 = 2;
   InX1_c1 =
Out2_c1;
   InX0_c1 =
Out0_c0;
   InA4_c1 = 0;
   InA3_c1 = 0;
   InD2_c1 = 0;
   InX4_c1 = 0;
   InX3_c1 = 0;
   InX6_c1 = 0;
%UPM c2
parameters
   InA2_c2 =
Out0_c1;
   InA1_c2 =
datax1;
   InA0_c2 =
Out0_c1;
   InX5_c2 = 0;
   InX2_c2 = 2;
   InX1_c2 =
Out2_c2;
   InX0_c2 =
Out0_c1;
   InA4_c2 = 0;
   InA3_c2 = 0;
   InD2_c2 = 0;
   InX4_c2 = 0;
   InX3_c2 = 0;
   InX6_c2 = 0;
 %UPM c2
parameters
   InA2_c3 =
Out0_c2;
   InA1_c3 =
datax1;
   InA0_c3 =
Out0_c2;
   InX5_c3 = 0;
   InX2_c3 = 2;
   InX1_c3 =
Out2_c3;
   InX0_c3 =
Out0_c2;
   InA4_c3 = 0;
   InA3_c3 = 0;
   InD2_c3 = 0;
   InX4_c3 = 0;
   InX3_c3 = 0;
   InX6_c3 = 0;
   %DIVISION
    output
   Output0 =
Out0_c3;
   Output1 = 0;
%%%%%%%%%%%%%
%%%Racine%%%%%
%%%%%%%%%%%%%
elseif
 (Opcode==6)
   %Control
 bits
   bit1 = 1;
   bit3_c0 = 1;
   %Add%Sub bits
    ad0_c0  = 1;
    ad1_c0  = 0;
    ad2_c0  = 1;
    ad3_c0  = 1;
    ad4_c0  = 1;
    ad5_c0  = 0;
%UPM c0
parameters
   InA2_c0 =
Samples1;
   InA1_c0 =
Samples;
   InA0_c0 =
0.5;
   InX5_c0 = 0;
   InX2_c0 = 1;
   InX1_c0 =
datax1;
   InX0_c0 =
Out0_c0;
   InA4_c0 = 0;
   InA3_c0 = 0;
   InD2_c0 = 0;
   InX4_c0 = 0;
   InX3_c0 = 0;
   InX6_c0 = 0;
   %Square
    output
   Output0 =
Out0_c3;
   Output1 = 0;
%%%Racine%%%%%
%
%%%%%%%%%%%%%
elseif
(Opcode==6)
  %Control bits
  bit1 = 1;
  bit3_c0 = 1;
  %Add%Sub bits
  ad0_c0  = 1;
  ad1_c0  = 0;
  ad2_c0  = 1;
  ad3_c0  = 1;
  ad4_c0  = 1;
  ad5_c0  = 0;

%Initialization
of
  %UPM c0
parameters
  InA2_c0 =
Samples1;
```

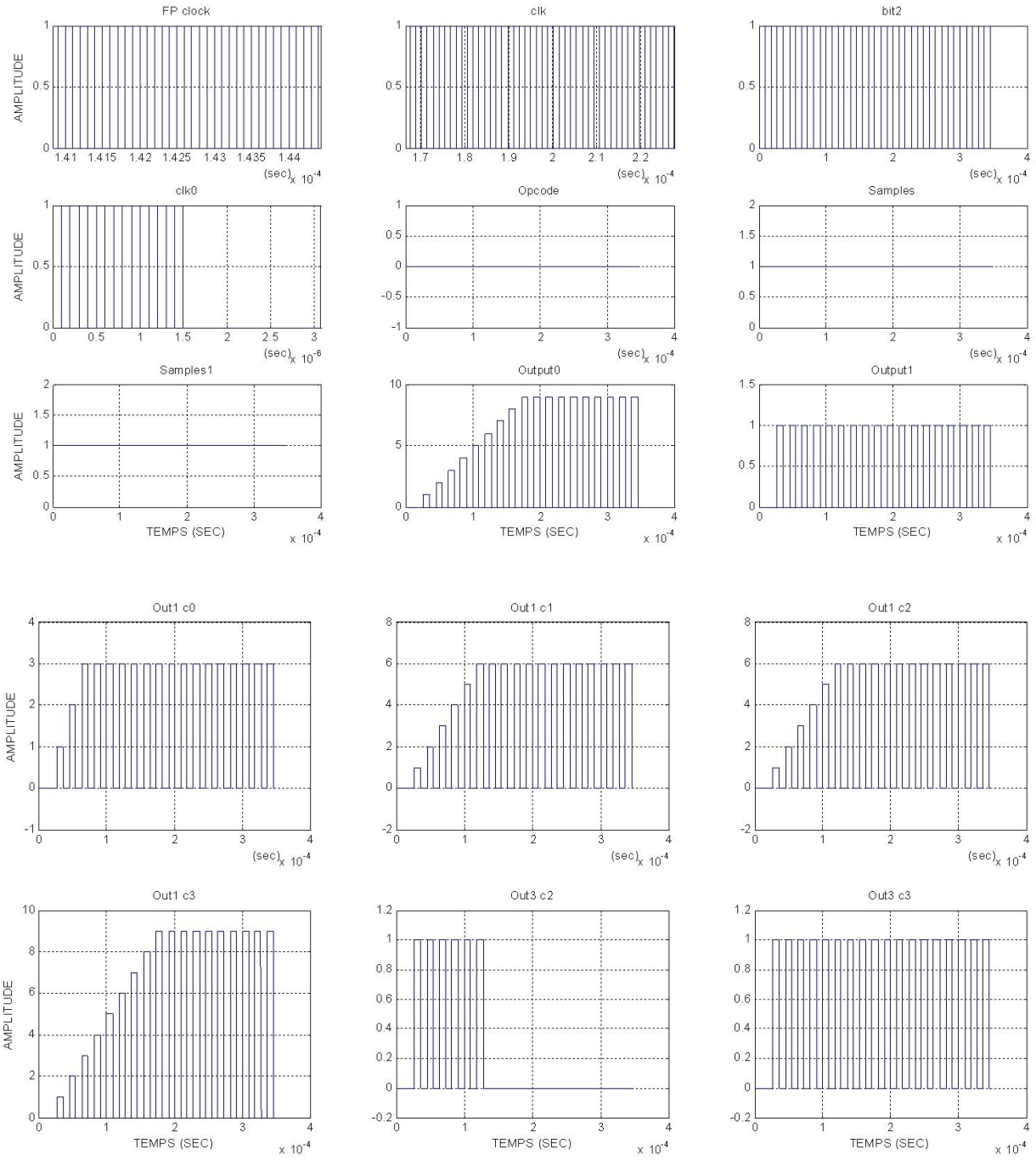
```
  InA1_c0 =
Samples;
  InA0_c0 =
0.5;
  InX5_c0 = 0;
  InX2_c0 = 1;
```

```
  InX1_c0 =
datax1;
  InX0_c0 =
Out0_c0;
  InA4_c0 = 0;
  InA3_c0 = 0;
  InD2_c0 = 0;
```

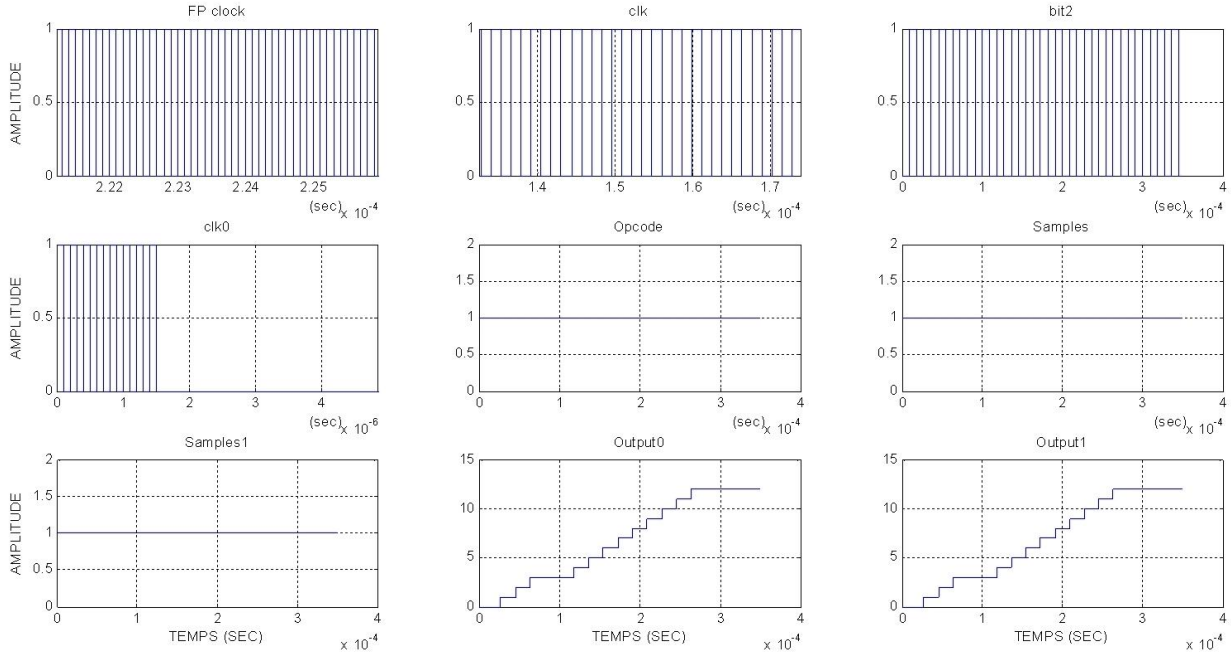
```
  InX4_c0 = 0;
  InX3_c0 = 0;
  InX6_c0 = 0;
  %Square
  %processor
output
```

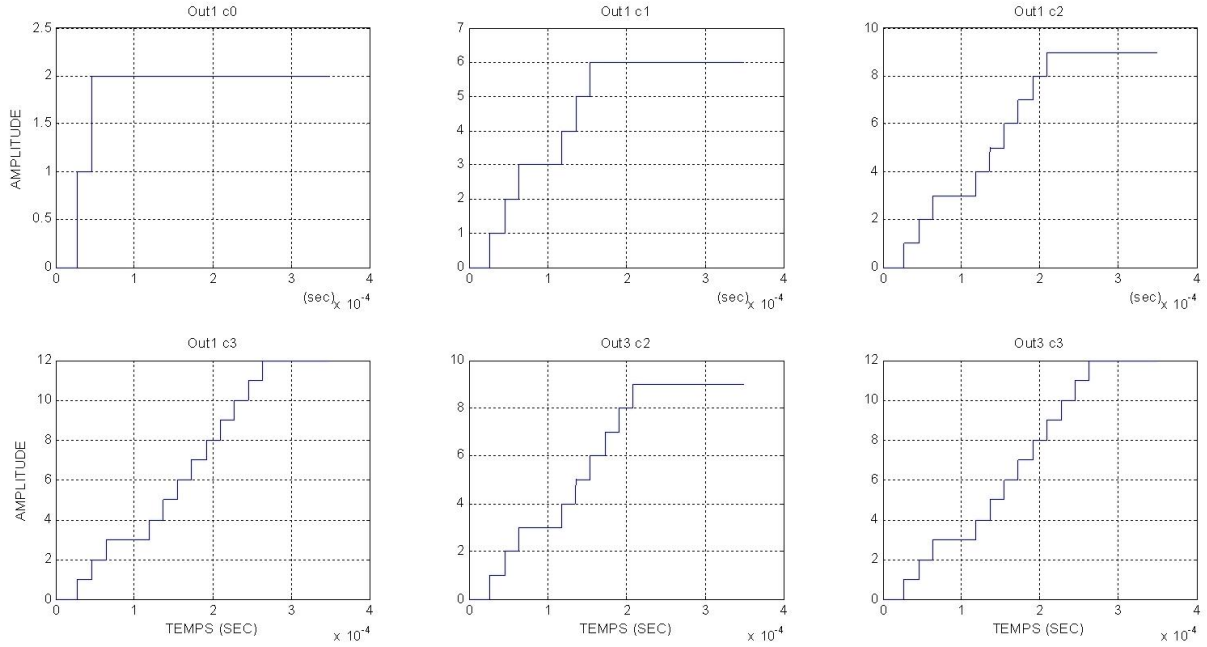
```
  Output0 =
Out2_c0;
  Output1 = 0;

   end
```

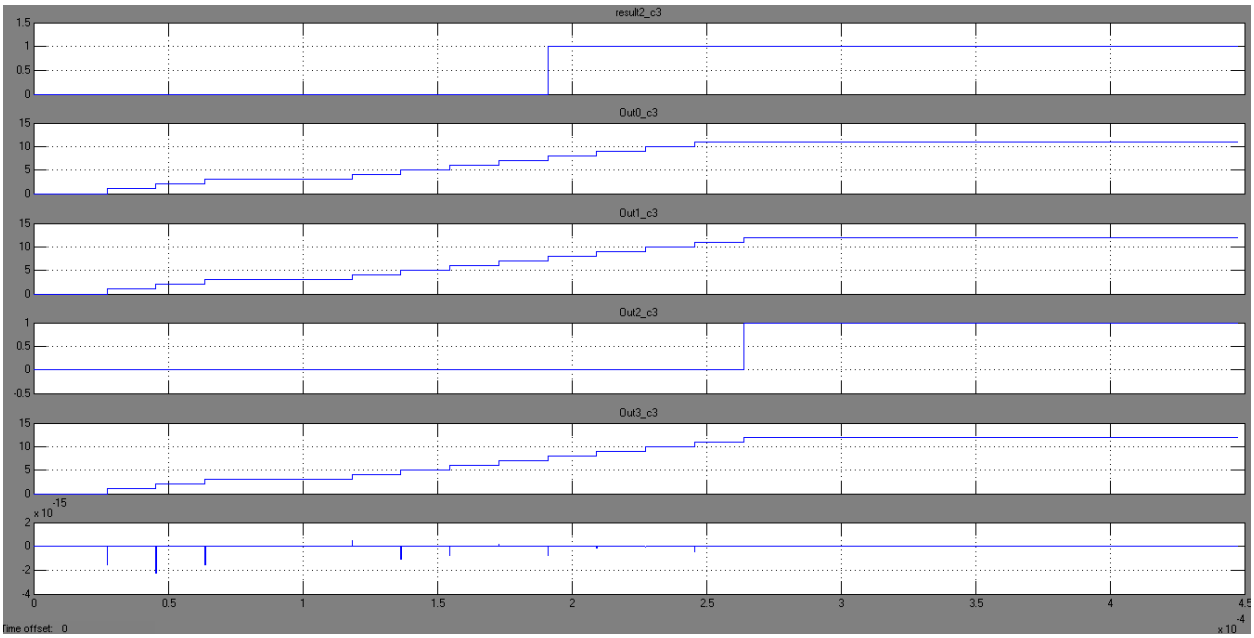
ANNEXE 45 – Simulation du filtrage FIR/IIR, *N*=8 sur Matlab.



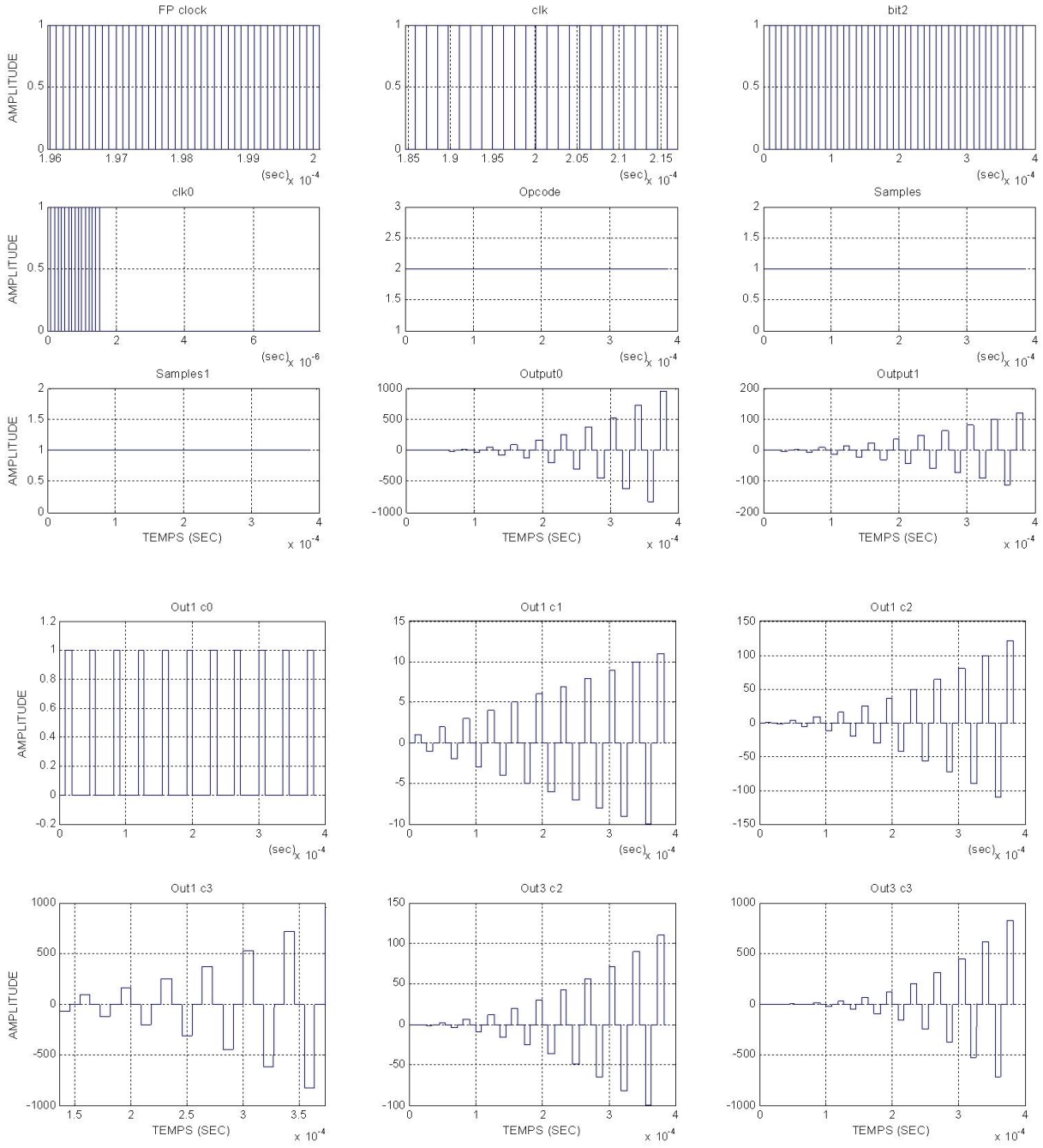
ANNEXE 46 – Simulation du filtrage FIR, *N*=12 sur Matlab.

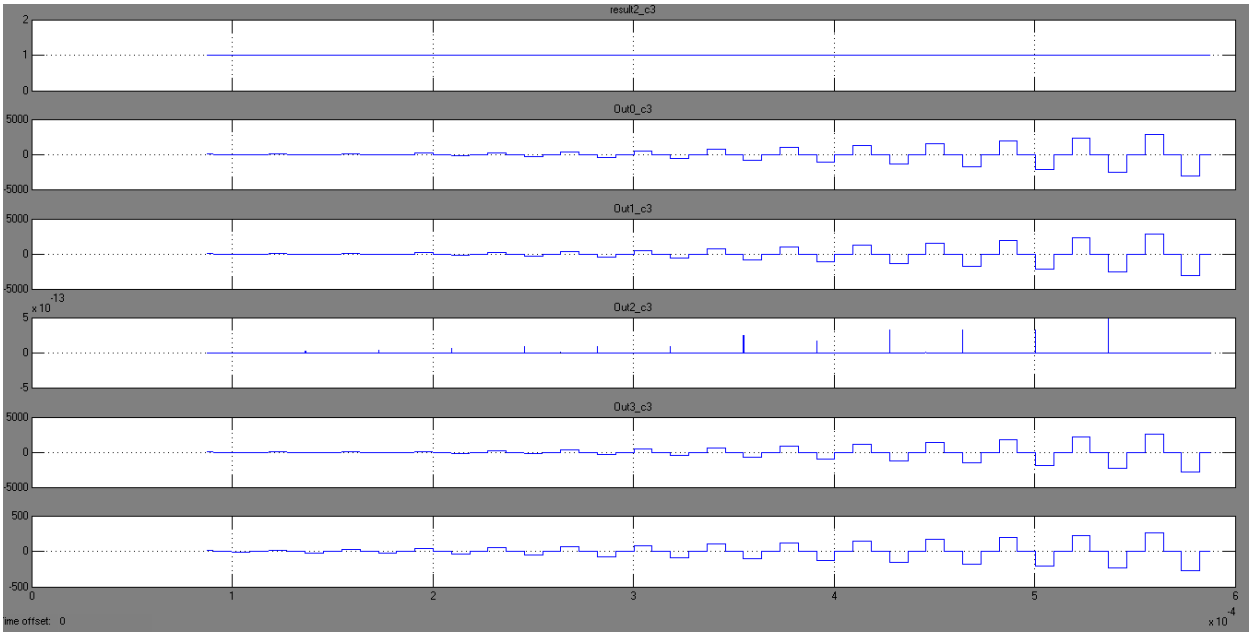



Out1 c0
Out1 c1
Out1 c2
Out1 c3
Out3 c2
Out3 c3
AMPLITUDE
TEMPS (SEC)
(sec)
x 10^-4

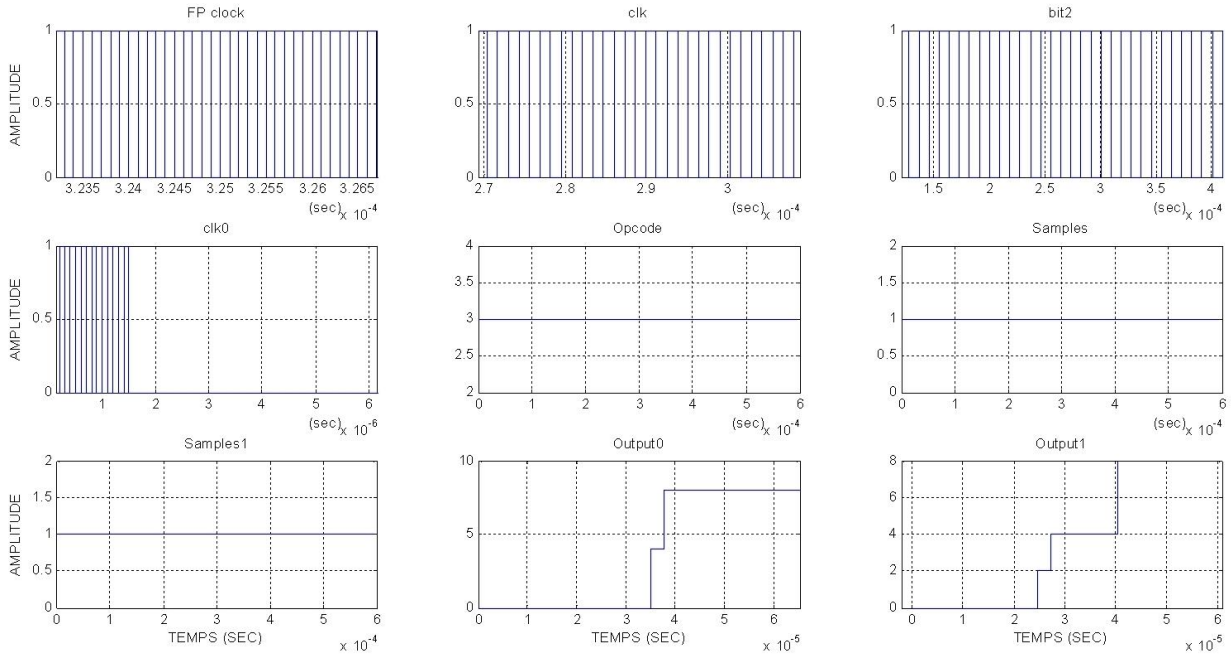

result2_c3
Out0_c3
Out1_c3
Out2_c3
Out3_c3
x 10^-15
time offset: 0
x 10^-4

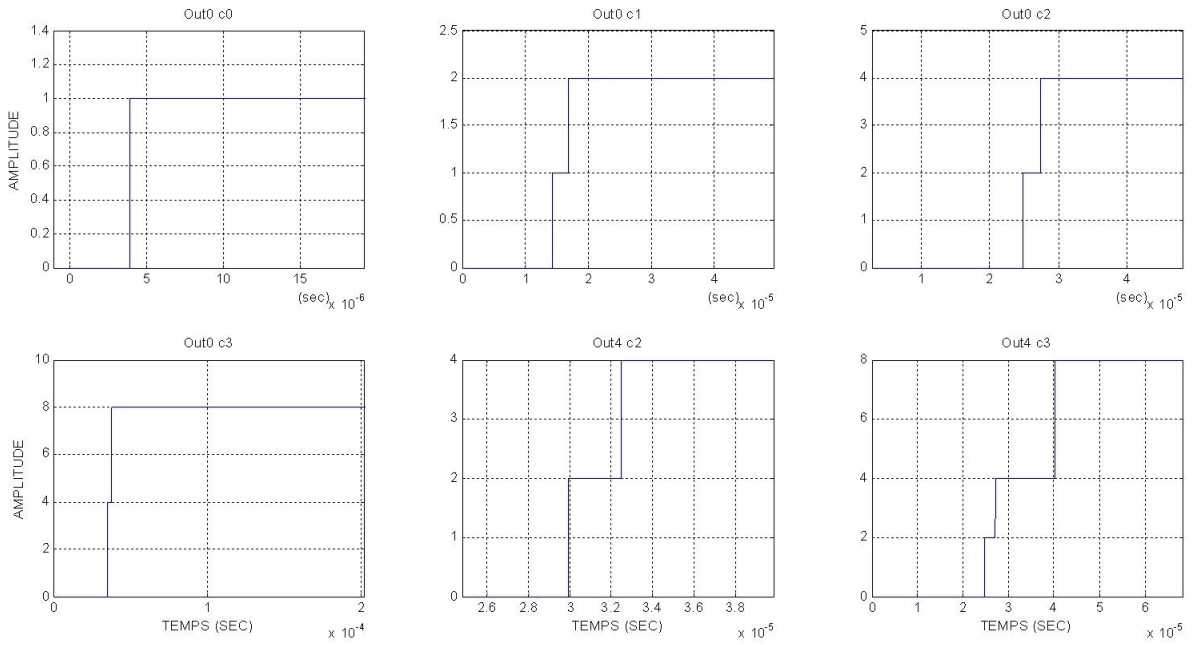
ANNEXE 47 – Simulation du filtre All-Pole, *s*=4 sur Matlab.

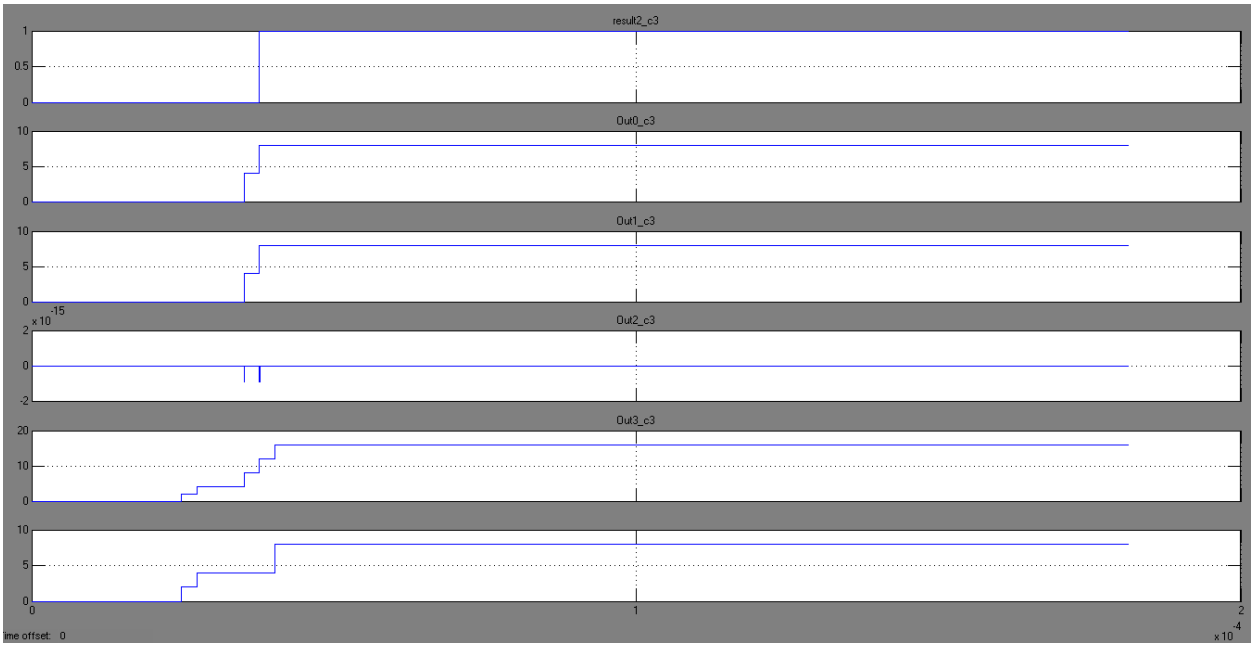




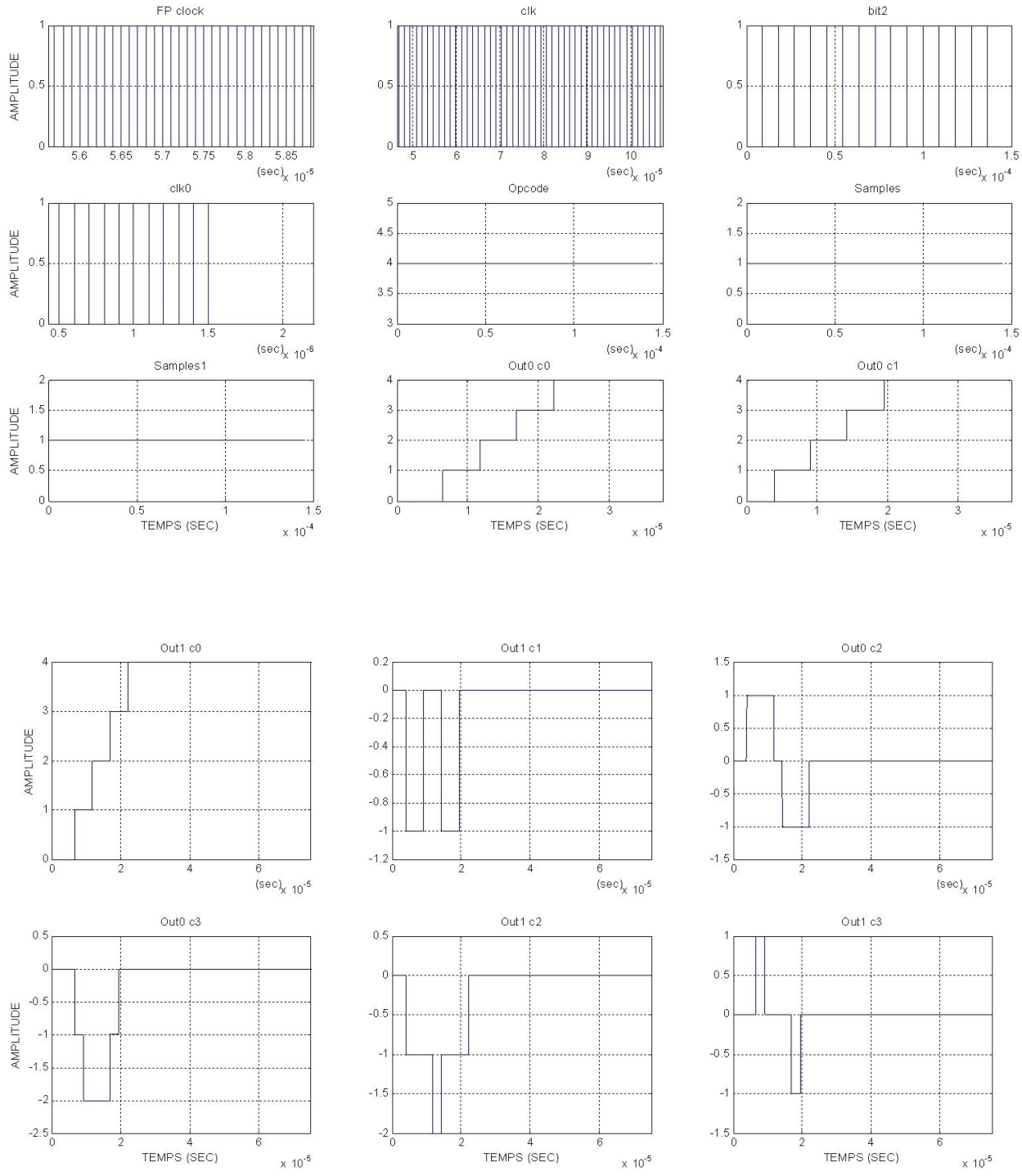
ANNEXE 48 – Simulation du filtre All-Zero, *s*=4 sur Matlab.

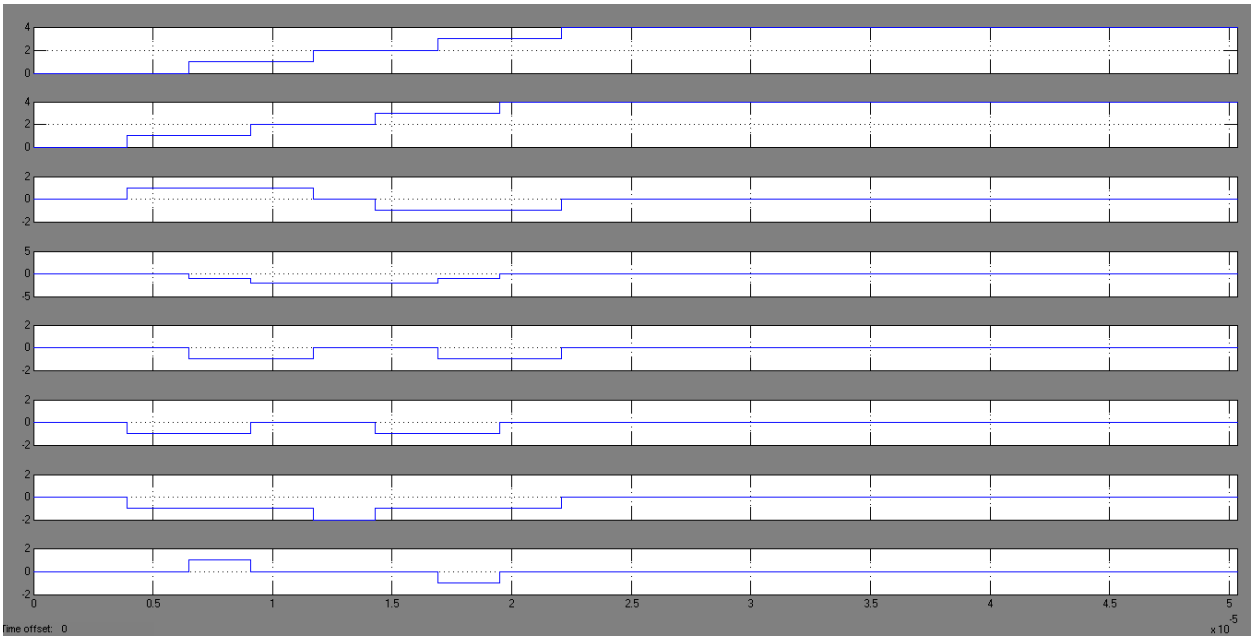



Out0 c0
Out0 c1
Out0 c2
Out0 c3
Out4 c2
Out4 c3
AMPLITUDE
TEMPS (SEC)
(sec)

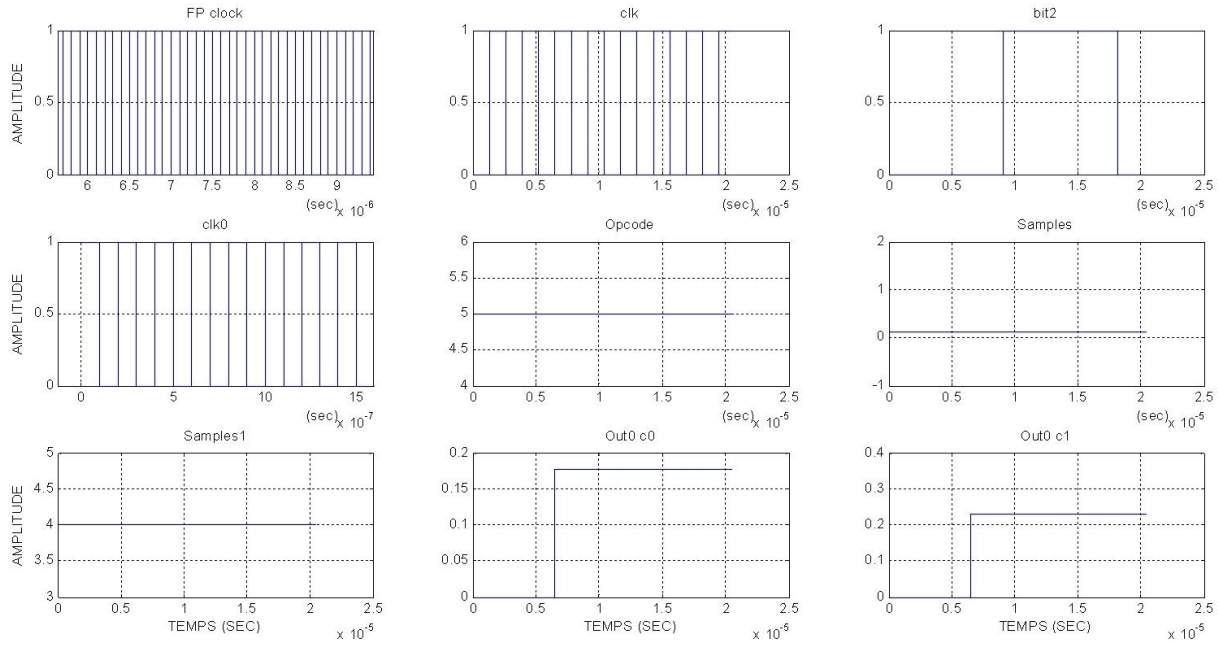

result2_c3
Out0_c3
Out1_c3
Out2_c3
Out3_c3

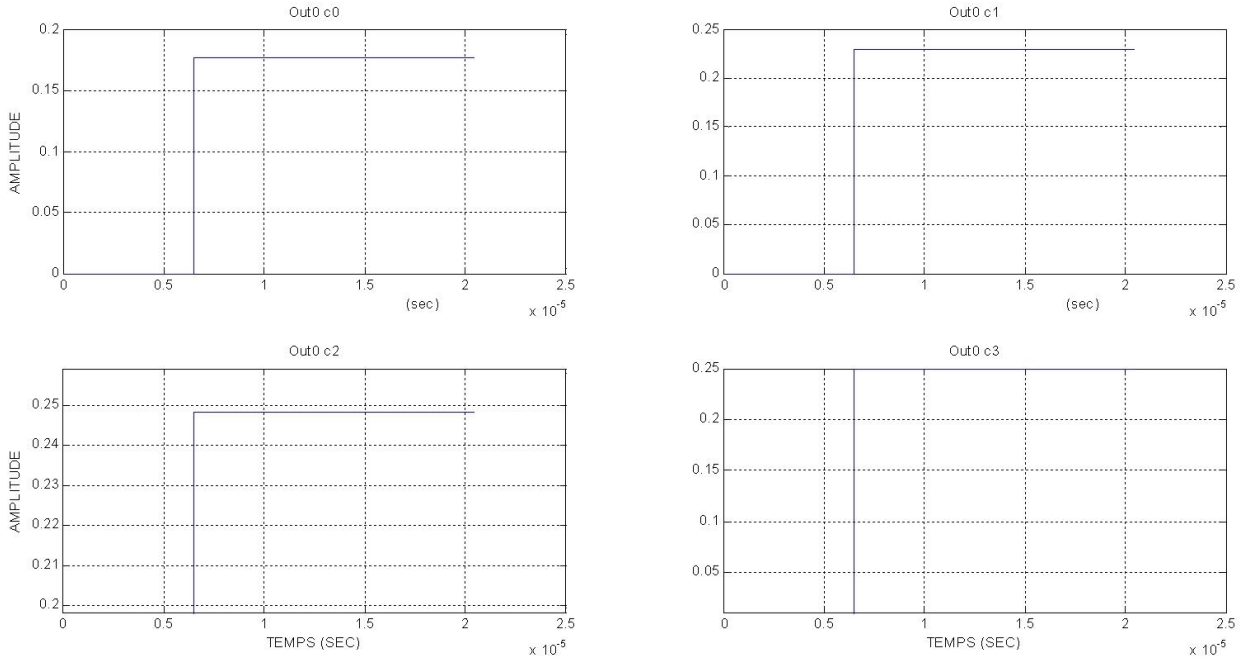
ANNEXE 49 – Simulation de la FFT radical-4 sur Matlab.

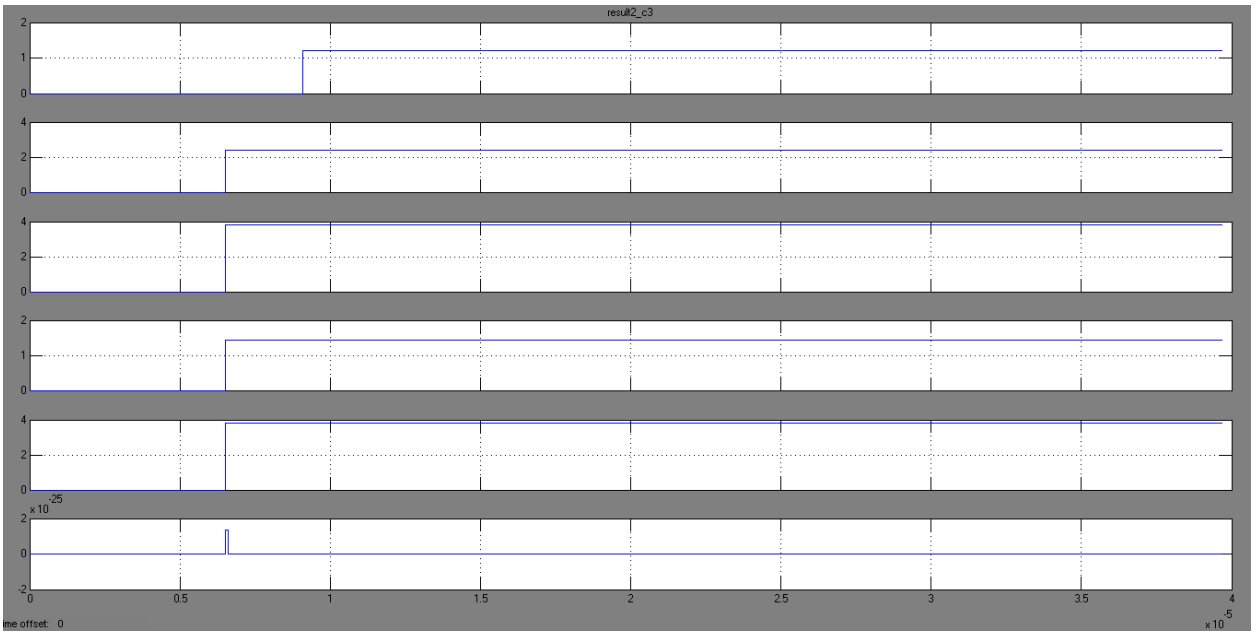




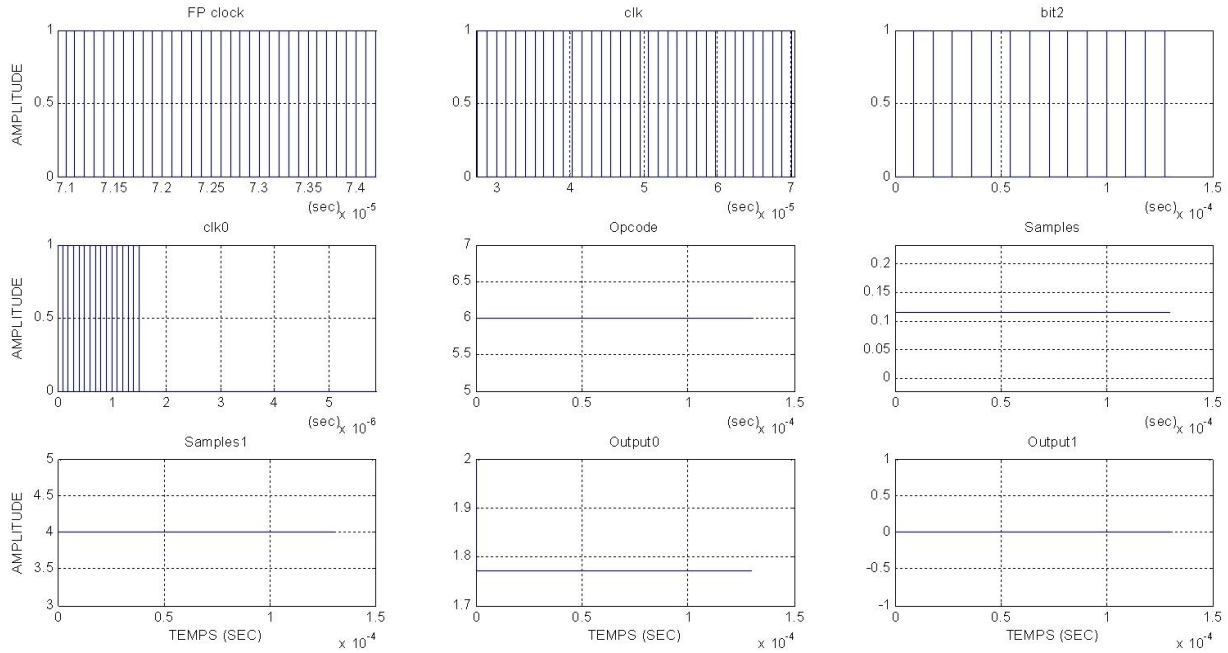
ANNEXE 50 – Simulation de la division sur Matlab.



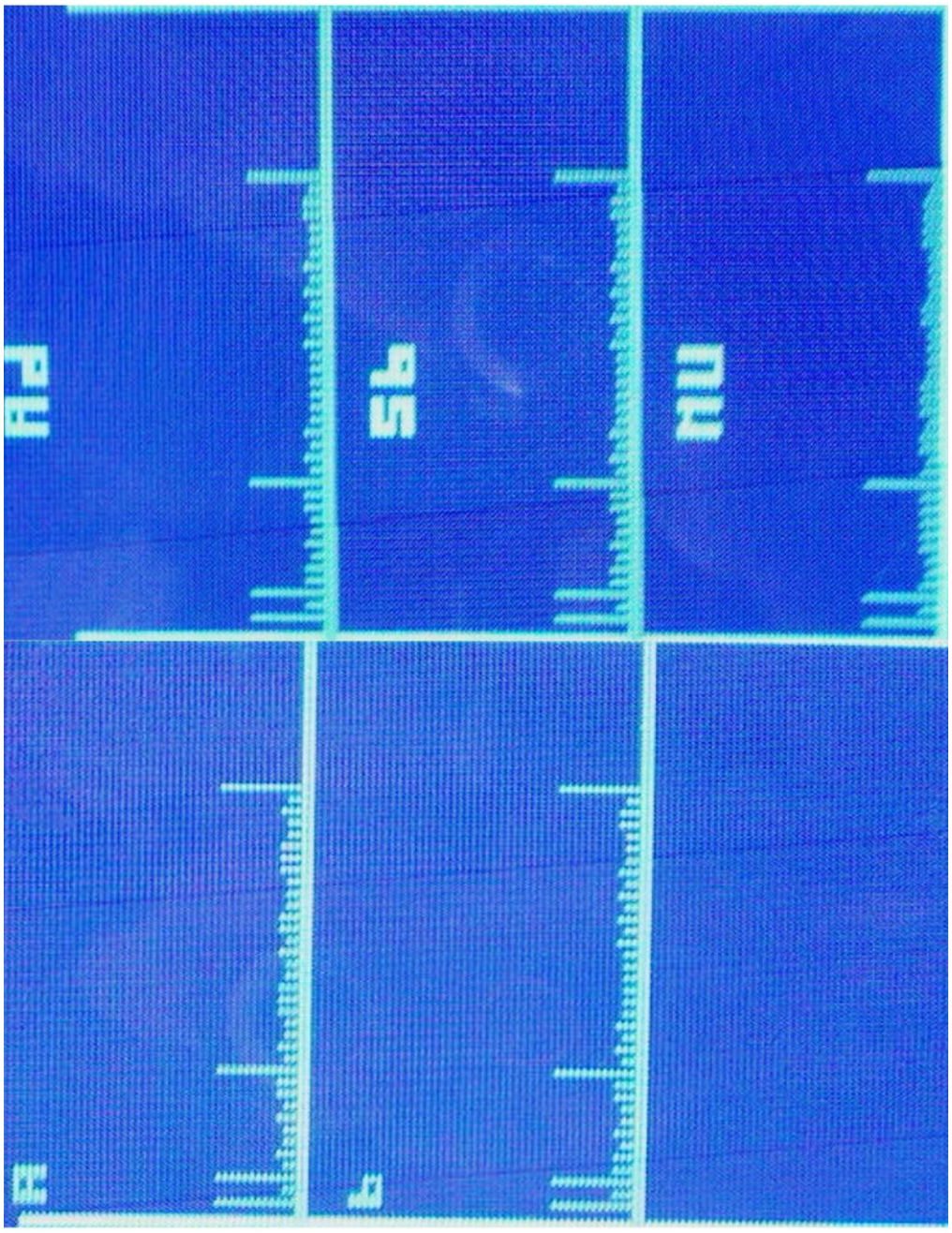

Out0 c0
AMPLITUDE
(sec)
x 10^-5
Out0 c1
(sec)
x 10^-5
Out0 c2
AMPLITUDE
TEMPS (SEC)
x 10^-5
Out0 c3
TEMPS (SEC)
x 10^-5


result2_c3
x 10^-25
x 10^-5

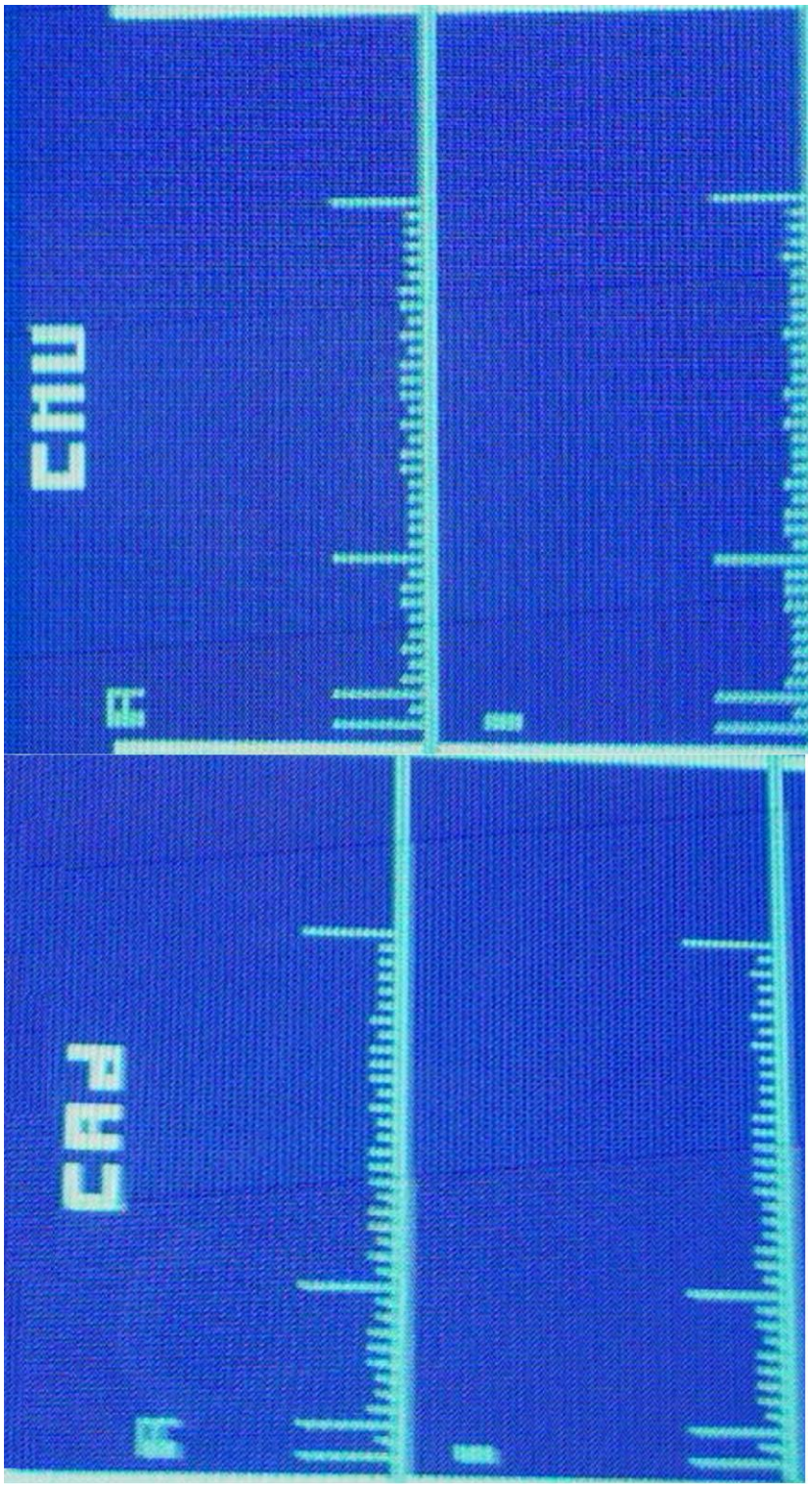
ANNEXE 51 – Simulation de la racine carrée sur Matlab.



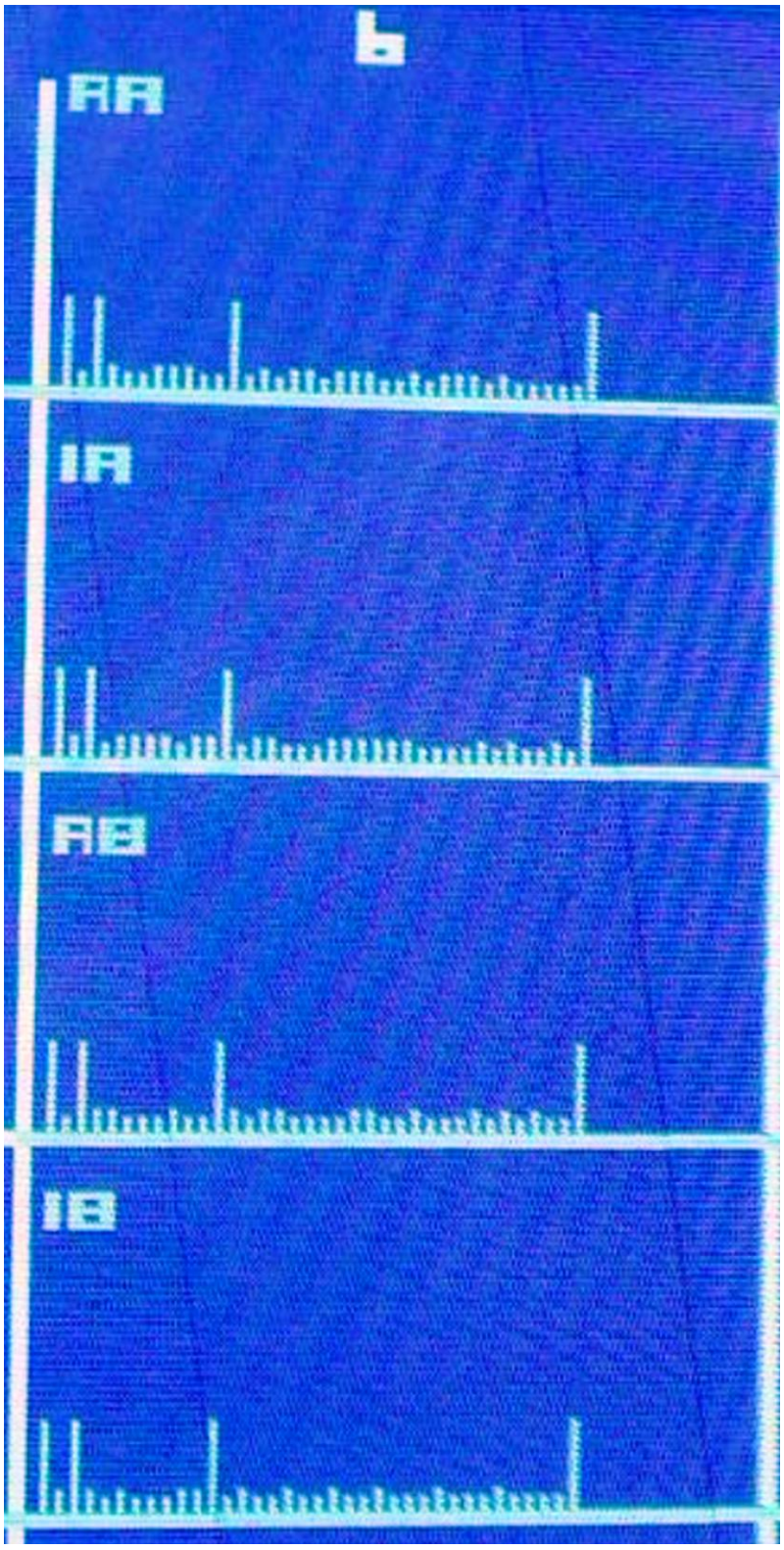
ANNEXE 52 – Test des opérations Point Flottant sur écran VGA.



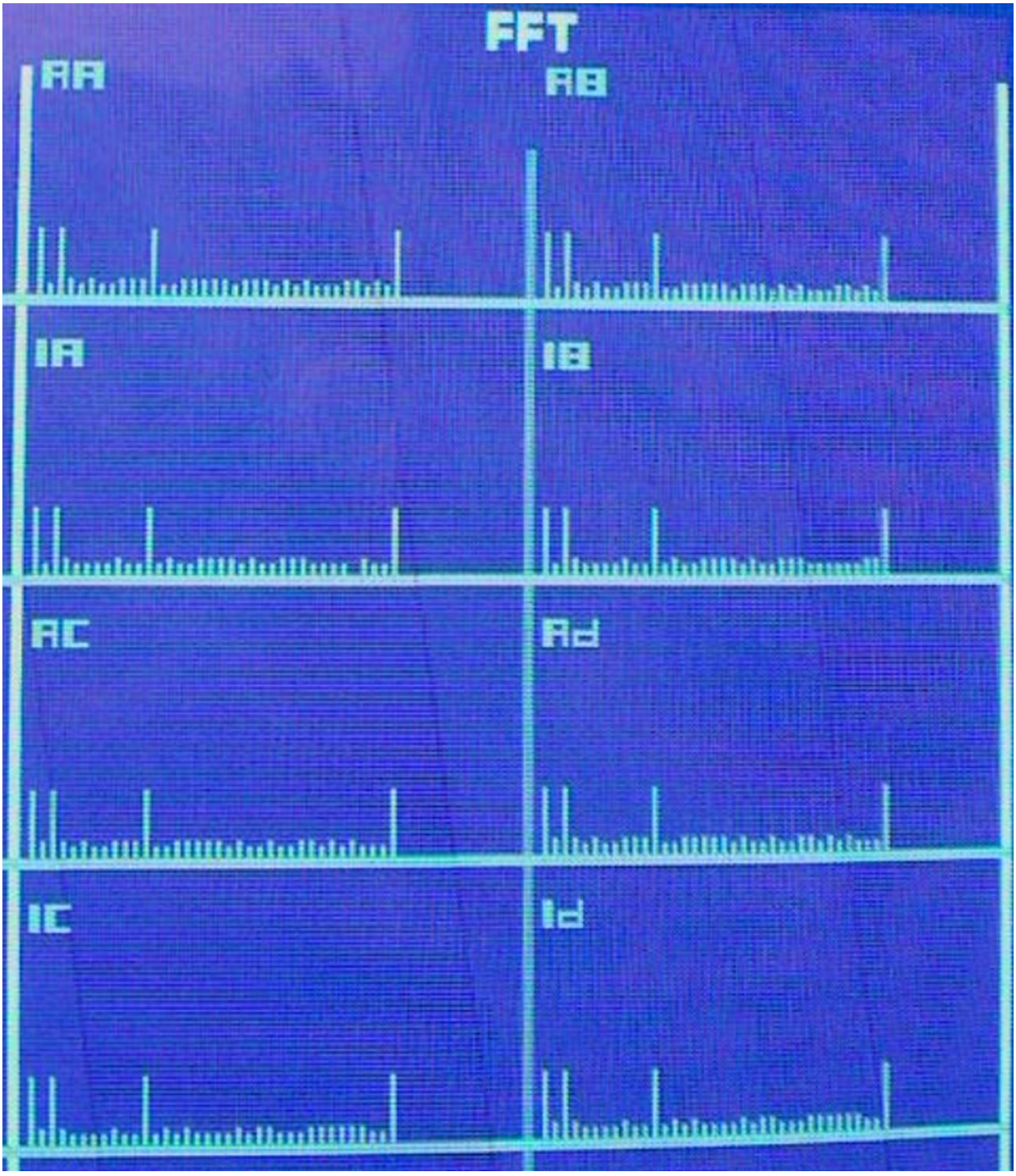
ANNEXE 53 – Test VGA de la multiplication et addition complexe.



ANNEXE 54 – Test VGA d’un operateur "*Butterfly*" radical 2.



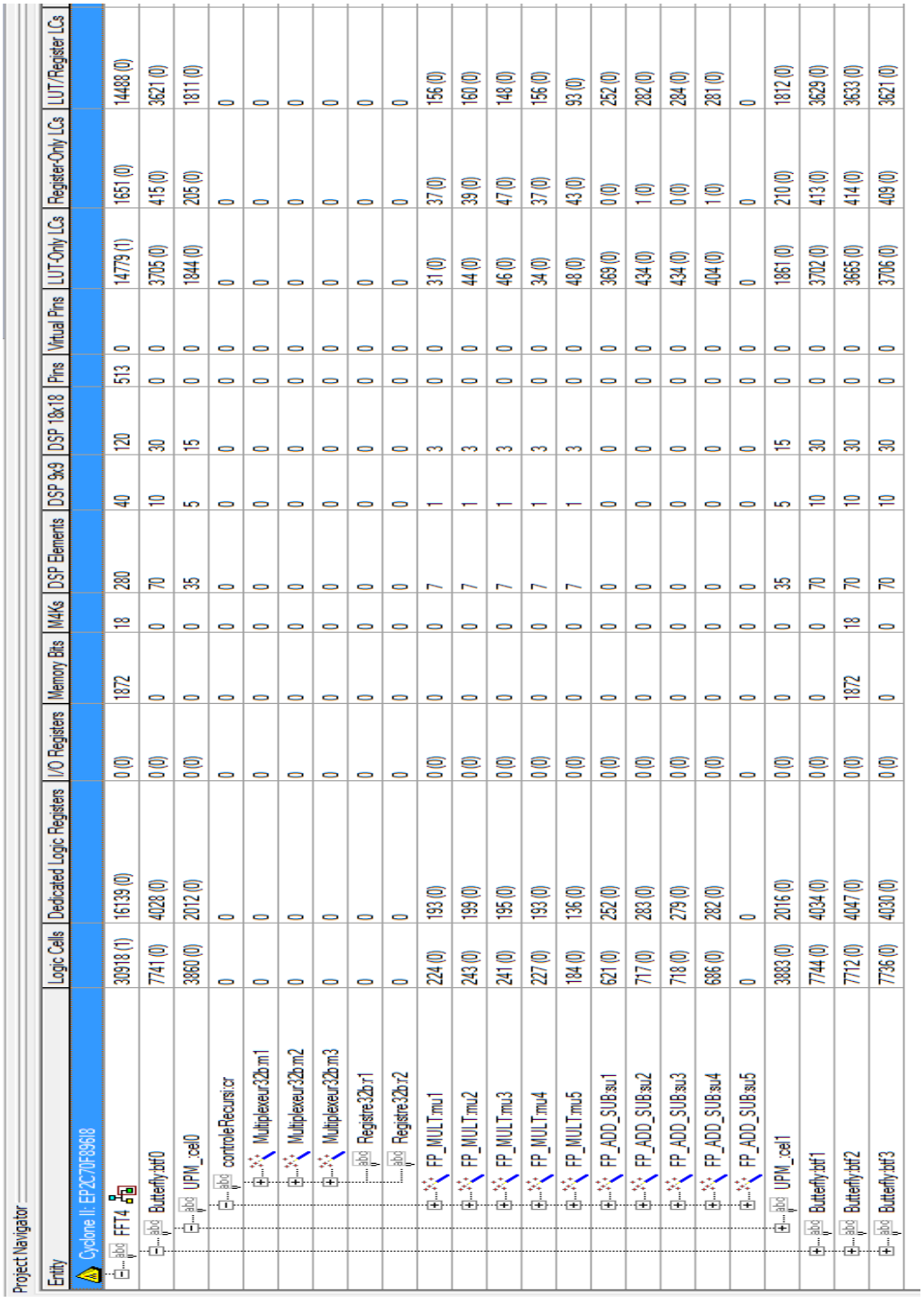
ANNEXE 55 – Test VGA d'une FFT, ordre N=4.



ANNEXE 56– Performances de vitesse des opérateurs point flottant sur FPGA

Famille du FPGA	**Opération point flottant**	**Délai de sortie (nombre de cycle d'horloge)**	**Fréquence max (*MHz*)**
Cyclone III (Altera)	multiplication	5	209
	addition	7	154
Stratix III (Altera)	multiplication	4	240
	addition	7	218
Stratix IV (Altera)	multiplication	4	274
	addition	7	228
Virtex 6 (Xilinx)	multiplication	-	429
	addition	-	403

ANNEXE 57–Ressources logiques utilisées par l'UPM



Project Navigator

Entity	Logic Cells	Dedicated Logic Registers	I/O Registers	Memory Bits	M4Ks	DSP Elements	DSP 9x9	DSP 18x18	Pins	Virtual Pins	LUT-Only LCs	Register-Only LCs	LUT/Register LCs
Cyclone II: EP2C70F896I8													
FFT4	30918 (1)	16139 (0)	0 (0)	1872	18	280	40	120	513	0	14779 (1)	1651 (0)	14488 (0)
Butterfly:btf0	7741 (0)	4028 (0)	0 (0)	0	0	70	10	30	0	0	3705 (0)	415 (0)	3621 (0)
UPM_cel0	3860 (0)	2012 (0)	0 (0)	0	0	35	5	15	0	0	1844 (0)	205 (0)	1811 (0)
controleRecursif:cr	0	0	0	0	0	0	0	0	0	0	0	0	0
Multiplexeur32b:m1	0	0	0	0	0	0	0	0	0	0	0	0	0
Multiplexeur32b:m2	0	0	0	0	0	0	0	0	0	0	0	0	0
Multiplexeur32b:m3	0	0	0	0	0	0	0	0	0	0	0	0	0
Registre32b:r1	0	0	0	0	0	0	0	0	0	0	0	0	0
Registre32b:r2	0	0	0	0	0	0	0	0	0	0	0	0	0
FP_MULT:mu1	224 (0)	193 (0)	0 (0)	0	0	7	1	3	0	0	31 (0)	37 (0)	156 (0)
FP_MULT:mu2	243 (0)	199 (0)	0 (0)	0	0	7	1	3	0	0	44 (0)	39 (0)	160 (0)
FP_MULT:mu3	241 (0)	195 (0)	0 (0)	0	0	7	1	3	0	0	46 (0)	47 (0)	148 (0)
FP_MULT:mu4	227 (0)	193 (0)	0 (0)	0	0	7	1	3	0	0	34 (0)	37 (0)	156 (0)
FP_MULT:mu5	184 (0)	136 (0)	0 (0)	0	0	7	1	3	0	0	48 (0)	43 (0)	93 (0)
FP_ADD_SUB:su1	621 (0)	252 (0)	0 (0)	0	0	0	0	0	0	0	369 (0)	0 (0)	252 (0)
FP_ADD_SUB:su2	717 (0)	283 (0)	0 (0)	0	0	0	0	0	0	0	434 (0)	1 (0)	282 (0)
FP_ADD_SUB:su3	718 (0)	279 (0)	0 (0)	0	0	0	0	0	0	0	434 (0)	0 (0)	284 (0)
FP_ADD_SUB:su4	686 (0)	282 (0)	0 (0)	0	0	0	0	0	0	0	404 (0)	1 (0)	281 (0)
FP_ADD_SUB:su5	0	0	0	0	0	0	0	0	0	0	0	0	0
UPM_cel1	3883 (0)	2016 (0)	0 (0)	0	0	35	5	15	0	0	1861 (0)	210 (0)	1812 (0)
Butterfly:btf1	7744 (0)	4034 (0)	0 (0)	0	0	70	10	30	0	0	3702 (0)	413 (0)	3629 (0)
Butterfly:btf2	7712 (0)	4047 (0)	0 (0)	1872	18	70	10	30	0	0	3665 (0)	414 (0)	3633 (0)
Butterfly:btf3	7736 (0)	4030 (0)	0 (0)	0	0	70	10	30	0	0	3706 (0)	409 (0)	3621 (0)

ANNEXE 58–Ressources utilisées par les matrices cellulaires 6x4

Resource	Usage	Resource	Usage
Total logic elements	133,132 / 198,464 (67 %)	M9Ks	58 / 891 (7 %)
-- Combinational with no register	66799	Total block memory bits	6,397 / 8,211,456 (< 1 %)
-- Register only	13443	Total block memory implementation bits	534,528 / 8,211,456 (7 %)
-- Combinational with a register	52890	Embedded Multiplier 9-bit elements	476 / 792 (60 %)
Logic element usage by number of LUT inputs		PLLs	0 / 4 (0 %)
-- 4 input functions	37391	Global clocks	4 / 20 (20 %)
-- 3 input functions	62826	Average interconnect usage (total/H/V)	43% / 42% / 45%
-- <=2 input functions	19472	Peak interconnect usage (total/H/V)	90% / 88% / 93%
-- Register only	13443	Total registers	66,333 / 200,539 (33 %)
Logic elements by mode		-- Dedicated logic registers	66,333 / 198,464 (33 %)
-- normal mode	85008	-- I/O registers	0 / 2,075 (0 %)
-- arithmetic mode	34681	I/O pins	140 / 430 (33 %)
Total LABs: partially or completely used	10,802 / 12,404 (87 %)	-- Clock pins	2 / 8 (25 %)
User inserted logic elements	0	-- Dedicated input pins	0 / 9 (0 %)
Virtual pins	0	Global signals	4